Specification-Based Verification and Testing of Open Distributed Systems

Arild B. Torjusen

Department of Informatics
Research group for precise modeling and analysis

Dissertation for the degree of Ph.D.
Submitted to the Faculty of Mathematics and Natural Sciences
University of Oslo

July 2010

Series of dissertations submitted to the
Faculty of Mathematics and Natural Sciences, University of Oslo
No. 1043

ISSN 1501-7710

Cover: Inger Sandved Anfinsen.
Printed in Norway: AIT Oslo AS.

Produced in co-operation with Unipub.
The thesis is produced by Unipub merely in connection with the
thesis defence. Kindly direct all inquiries regarding the thesis to the copyright
holder or the unit which grants the doctorate.

Summary

The motivation for this dissertation is to increase the usefulness of Creol as a modeling language for open distributed systems and through this contribute to the overall goal of verification and testing of open distributed systems. We develop methods for tool-based testing and verification of Creol models, by introducing two different formal languages for specification of Creol components using behavioral interfaces. The formalisms lead to two different ways to use these specifications to build frameworks and tools for automatic testing of Creol models.

Creol is a modeling language that is specifically designed for modeling open distributed systems with asynchronous communication. The basic programming paradigm of Creol is object orientation. The syntax of Creol is quite similar to a programming language, but Creol abstracts certain properties; thus some aspects of the system may remain undetermined in the model. In this way we get models that are more abstract than the full system, but that may be structurally quite similar to the systems and also quite complex. This again makes it necessary to ensure that also the model conform to the intended behavior of the system.

By exploiting capabilities of the rewriting logic execution platform Maude—such as metalevel rewriting, efficient state exploration, and rewriting modulo equational attributes (associativity and commutativity)—we achieve efficient methods for assume-guarantee style specification-based testing and model checking of Creol components. The methods we have developed address the additional challenges for verification and testing that arise from the non-determinism of the model. We have implemented the methods as testing frameworks in rewriting logic together with examples. We have experimented with the frameworks to evaluate the testing methods. The experiments show that both methods are useful for building frameworks for automatic testing of Creol model components. Thus the main result of the dissertation is tool-supported methods for verification of Creol models of open distributed systems, and consequently methods for specification-based verification and testing of open distributed systems.

Preface

I wish to thank my supervisors Einar Broch Johnsen, Olaf Owe, and Martin Steffen for excellent cooperation, support, and guidance throughout the period of my Ph.D. scholarship. In particular, I wish to thank Martin Steffen who joined the supervisor team at a later stage but who has contributed greatly to my understanding of theoretical computer science in general and the subject of this dissertation in particular.

I would also like to thank everyone in the PMA group for interesting discussions, for valuable feedback, and for being very good colleagues, and in particular Ingrid and Johan for a good time while sharing office in the early years of this period. Thanks to Immo Grabe, Gerardo Schneider, and Marcel Kyas for collaboration on papers, and to Andreas Grüner and Rudolf Schlatte for sharing their insight into testing. I also thank the anonymous referees for constructive criticism to the papers. I am grateful to readers of earlier drafts of the dissertation and of the research papers. Many have contributed, but I wish in particular to mention Bjarte M. Østvold, whose criticism contributed significantly to improve the final version of the dissertation.

Above all, I thank my wife Line for her everlasting support, love, and patience, and my two lovely daughters Karoline and Cathinka, who were both born in this period. You make my world a brighter place every day.

The main support for this work has been through the NFR project *CREOL: A formal framework for reflective component modeling.* Part of the work was supported by the EU-project IST-33826 *Credo: Modeling and analysis of evolutionary structures for distributed services*, the German-Norwegian DAAD-NWO exchange project *Avabi: Automated validation for behavioral interfaces of asynchronous active objects*, and the NFR project *Connect: Active Behavioral Interfaces for Seamless Network Integration*.

As is customary at the Faculty of Mathematics and Natural Sciences at the University of Oslo, the dissertation consists of a collection of research papers preceded by an introductory overview. The overview in Part I introduces the research problems that we address and the research goals we aim to reach, I give some relevant background and evaluate our contribution. In Part II the five research papers are included as they were originally published.

Contents

Part I

Overview

Chapter 1

Introduction

Ensuring software reliability is vital in a world where we increasingly rely on the correct functioning of computers and their software. Standard examples of systems where reliability is of particular importance are applications for air traffic control, power plants, oil and gas exploration, or space craft missions. In such systems, software failures may have dire consequences. Failures in systems for e-commerce, e.g., in software for validation of credit card transactions, may *seem* less dramatic; the correct functioning of these however is yet as important as this affects us all on a daily basis. Another example is electronic voting, which may promote democracy by making the voting process more accessible and efficient, but where reliability of the system is of vital importance for the participants' confidence in the results. Software may also function correctly under normal circumstances but have inherent weaknesses that might be exploited by malicious third parties. This question of security is of the utmost importance as critical systems are in constant risk of attacks. To uphold the defense against such attacks is very costly.

This dissertation addresses the question of how to ensure correctness and reliability of software in the context of *open distributed systems*. A program is *correct* if it behaves as it was intended to, or more technically, if it implements its specification. In *specification-based* testing, one seeks to assert correctness by establishing a relation between the specification and the implementation. In the context of software engineering, *reliability* may be defined as the ability of a system to perform its required functions under stated conditions for a specified period of time [Std90]. *Distributed* systems are systems that consist of possibly heterogeneous IT systems that are interconnected in some way. For large-scale applications, this would typically be via the internet, but a system might just as well be distributed over a local area network. In this context, correctness of a system means not only the correctness of a program running on one computer, but also the correctness of a system as a whole where the system is distributed over different locations in a network and even executing on different architectures.

An *open* system is a system whose environment is not fixed. Typically, the environment (as e.g. the internet) may be changing and the system needs to be able to evolve and adapt to these changing requirements. Open systems should provide portability and interoperability. *Portability* implies that components should be able to execute and function properly on different nodes in a network without modification. *Interoperability* is the ability of a component to interact properly with components on other nodes, even with little or no knowledge of these units [Int95]. This openness is also reflected in the fact that a distributed system providing some service is often designed to run continuously and both modification of existing software and addition of new software must take place while the system is running. This means that the components should be able to adapt to

and function properly in unknown environments. Consequently, when developing, modeling, or reasoning about such components, one should only make minimal assumptions about the environment. Accordingly, an *open environment* is an environment in which various other software units exist, and little or no information about these units is available.

Open distributed systems are important. Via the internet, people may access bank services, participate in commerce or social networking, and interact with municipal and central authorities on matters regarding welfare, tax, and health services. The internet is a central part of the society infrastructure in many countries. Furthermore, the internet is evolving; with the availability of small inexpensive processing devices, not only traditional personal computers and servers, but also all kinds of physical objects with processing capabilities may interconnect, and information-processing distributed systems thus become even more integrated into our daily lives. This development is known as *ubiquitous* or *pervasive* computing, and envisions an *Internet of Things*.

In general, one does not have access to the implementation details of the components that the distributed system consists of. However, some knowledge of the components is often given in forms of abstract specifications of their behavior. With regard to the behavior of the overall system, components with different implementations may be interchanged if their behavioral specifications are compatible. In other words, we may take a *black-box* view on components, where we do not consider the internal details but only the *observable* behavior of the components.

This external view on software components and the openness of the environment with which they interact, lead to a compositional view on such systems. One does not have access to the internal details or code of the components that form the system; also, such systems are dynamic in the sense that which components are available in the system, changes over time. For reasoning about and for verification of open systems it is sensible to consider each component in isolation without making too strong assumptions about the environment with which it interacts; therefore, an *assumption-commitment* [MC81] or *assume-guarantee* [AL93] view on components is natural. The specification of a component will typically express that given the assumption that the environment in which the component executes behaves in a certain way, the component commits to or guarantees a certain behavior. This compositional approach is useful in order to tackle the complexity of such systems.

An influential current design paradigm for system development is *service orientation*, which aims at service-oriented architectures (SOA) [SOA]. It is a high-level approach where services are seen as autonomous entities that can be composed into larger solutions. While service orientation is an implementation-neutral concept, it is commonly associated with *web services*, which is but one possible implementation. Characteristic features of web services are that they support *extensibility*, and that they can be combined in loosely coupled ways to achieve complex operations.

Loose coupling points to a high degree of independence of the constituent parts of the systems and indicates that the parts communicate via message passing and not via other, more tightly coupled ways, as e.g., via a shared storage. Using *asynchronous* message passing further contributes to such independence by decoupling caller and callee. This allows a caller to proceed with other tasks while the callee processes the request, instead of idly waiting which would be the case if *synchronous* message passing were used. Open distributed systems are typically loosely coupled; therefore, an asynchronous communication model is advantageous as a basis for modeling such systems.

For complex systems, *modeling* allows for simplification of the system by focusing on selected properties while abstracting from others. Since open distributed systems typically are complex, it is useful to study an abstraction or a behavioral model of the system for verifying behavior. It is

necessary to use a language for models and specifications that is sufficiently rigid to describe the behavior of the system in a precise manner, and sufficiently formal to enable *automatic* and tool-based analysis, testing, and verification techniques.

1.1 Research goals

The subject of this dissertation is open distributed system, and the main research goal is:

Overall goal. *Specification-based verification and testing of open distributed systems.*

We explore open distributed systems at the model level and in particular through the Creol [JOY06, Cre07] modeling language. When modeling open distributed systems, it is an advantage that the model reflects the nature of the real system with regard to distribution, concurrency, and asynchronous communication. In this respect, Creol is a suitable framework for executable modeling of open distributed systems, as argued below.

1.1.1 Creol

Creol is a high-level executable object-oriented modeling language that specifically targets open distributed systems. In Creol we have a formal language that supports compositional reasoning through encapsulation of objects and activity (execution threads), that is sufficiently rigorous, and is *executable* which makes simulation of behavior and automatic testing and verification possible.

Creol provides high-level programming constructs, which unite object-orientation and distribution in a natural way. The concept of *active objects* is central in Creol, conceptually an object encapsulates an execution thread; by means of processor release points, objects may dynamically change between active and passive (reactive) behavior [JO07]. All object interaction is through *asynchronous method calls*; this imposes a structure and discipline to the message passing, yet avoids the tight coupling that would result from a synchronous model. The operational semantics of Creol is defined in rewriting logic [Mes92], and Creol models are directly executable in the rewriting logic system Maude [CDE$^+$02], which provides an interpreter and analysis platform for the models.

Creol has a well-developed proof theory that supports compositional reasoning [DJO07]. Creol is a type-safe language; a type system for Creol is introduced in [JOY06], where type checking of asynchronous method calls is based on a type and effect system. The authors show that runtime type errors do not occur for well-typed programs. Modular reasoning and verification techniques for Creol are studied by Dovland in [Dov09]. In particular, a compositional proof system for Creol is given; Dovland shows how specifications of a complete system can be derived by composing interface level specifications of the individual objects of the system.

The similarity of Creol and a more low-level object-oriented imperative *programming* language, together with Creol's expressiveness, allow for models that are structurally close to the programs being modeled. By using Creol for modeling a program, we get an executable model, which gives possibilities for rapid prototyping and experimenting during model development.

1.1.2 Creol-specific research goals

Creol models are "programmer friendly"; they are easy to develop using programming language like constructs. The abstractions of the modeling language resemble those of an imperative programming language, but leave out low-level details. This leads to a high level of abstraction in the model but introduces additional non-determinism. Together with the fact that these models might be large and complex, this makes prediction of model behavior hard. Consequently, using Creol for modeling leads to models whose behavior is not obvious and which might need verification against behavior specifications. One might object to this approach by pointing out that the point with modeling is to create a model that gives a precise description of the behavior of the system, so why use Creol if the models are unpredictable? An answer to that is that Creol models are useful for system development and analysis.

Creol models are useful both top-down, going from model to implementation, and bottom-up, going from code to model. Top-down, the Creol model is a pre-version of the system that is to be implemented. As an executable prototype, the model can be used for simulation and visualization of the behavior of the system being developed and as a basis for later refinement to production code. In the other direction, a Creol model is useful in reverse engineering of existing complex programs [AGSS08]. The existing program can be reconstructed in a more abstract manner as a Creol model. By experimenting with the model and executing it, one may gain understanding of the program.

The motivation for this dissertation is to increase the usefulness of Creol as a modeling language for open distributed systems, and through this contribute to the overall goal of verification and testing of open distributed systems. In order to use Creol for modeling complex systems, we need better methods for testing Creol models. To achieve efficient testing of Creol models, which can be large and complex, manual testing is too limited, and tool support and automatic testing are called for. For testing complex systems there is a recent trend towards employing formal methods in order to facilitate test automation and tool support, and *model-based* testing is an approach to testing that is considered to be promising, see e.g. [H+09, HBH08, VCG+08, BJK+05]. Following this trend, we have chosen to focus on specification-based verification[1] and *testing* of Creol models. The main Creol-specific research goal is therefore:

Goal 1. *Tool-supported methods for verification and testing of Creol models of open distributed systems.*

The purpose of testing in our setting is to make sure that the Creol models that we develop actually have the behavior that was intended. The process of determining through testing whether an implementation satisfies a prescription of what the system should do is known as specification-based testing, or as *conformance testing* [Tre99]. The prescription of the behavior of a system is the *specification* of the system. There are two elements in this picture:

1. the *specification language*, and
2. the process of *testing* for conformance between the specification and the implementation.

Hence, we divide Goal 1 into sub-goals where the first two address the specification language. The first sub-goal is:

[1] Verification is not understood in the narrow sense of Hoare logic verification, but as the process of checking that a system meets its specification. *Testing* is therefore a verification technique. For a more extensive comment on terminology, see Chapter 4.

Goal 1.1. *A formal language for interface specification of the behavior of Creol components.*

Note that (1) The goal is *formal* descriptions of behavior, this to enable executability and tool support. (2) As explained above we assume only restricted knowledge of the implementation details of the components, and accordingly we will focus only on behavior observable at the interface of the components. (3) We take a compositional view and consider one component at a time, as a means to tackle the potential complexity of the models. Besides laying the foundation for conformance testing, this first sub-goal also serves in itself to contribute to make Creol a better modeling language by providing a language for interface level description of Creol components.

As mentioned above, the systems that we model are typically distributed heterogeneous systems consisting of quite different kinds of nodes, where little knowledge about the implementation details of the nodes can be assumed. However, for nodes to be able to interconnect, and for services to be able to work together, some interface descriptions are necessary. Such interface descriptions can be at the level of data, or at the level of control. In other words, in terms of what kind of data a node expects and returns, or in terms of what behavior a node might exhibit. We intend to look at interface specifications at both levels. For behavior, we will look at ways to specify components in terms of the *traces* observable at their interface. For data, we will consider how to specify Creol data structures using a language independent data format like XML. XML is widely used for data exchange between nodes in systems or networks where the nodes are independent and do not share a common interface. The second sub-goal is therefore:

Goal 1.2. *A data-based interface specification language for Creol.*

With regard to the testing process itself, we formulate a third sub-goal, which is:

Goal 1.3. *Methods for verification of Creol models. The methods should:*

1. *be automatic and supported by tools,*
2. *be compositional, and*
3. *tackle non-determinism.*

We have argued for the need for tool-support above. The methods for verification should reflect the nature of the systems that we test; we target open systems, which in their nature are component based, and therefore the testing methods should reflect this by being *compositional*, i.e., such that results from testing the behavior of components can be used to reach conclusions about the behavior of the composed system. In an open system there will be non-determinism caused by distribution; the asynchronous communication model is also a source of non-determinism. This leads to an increased state space and is a challenge for verification, which we have to address.

As mentioned above model-based testing is a promising approach to testing, and as milestones towards Goal 1.3 stated above we identify the following further sub-goals:

Goal 1.3.1. *Model-based testing of Creol models.*

To reach this goal we need (1) A conformance relation that can be used in our setting of Creol models. (2) A methodology for using this relation for testing. We will look into how we can adapt existing approaches, and a natural starting point is existing theories for model-based testing, which we will come back to in Section 4.1.

As we aim for tool-supported methods, we will also *implement* the tools and frameworks necessary for actually doing testing of Creol models based on the theoretical approach. The final sub-goal is therefore:

Goal 1.3.2. *Executable frameworks, including development tools, for model-based testing of Creol models.*

The execution platform for the Creol language is the rewriting logic system Maude. The testing frameworks and tools will be designed to run on the same platform. The implemented solutions are presented in Chapter 5.

1.1.3 Contributions

We discuss the contributions of the dissertation in detail below; in brief, the contributions are the following.

For *Goal 1.1*: We have developed two different formalisms for behavioral specification of Creol components. Both formalisms give assumption-commitment style specifications of Creol component behavior. The first is in the form of abstract logical predicates over the observable history of a component. The second is a more explicit trace language where a specification is given as a sequence of possible communication events.

For *Goal 1.2*: We describe how to extend Creol with XML, and we do the first steps of an implementation of XML support for Creol. By introducing XML into Creol and enhancing Creol with XML data types as well as devices for XML type checking using XML schema, we may specify Creol component interfaces in terms of data. This work is reported in Paper #3.

For *Goal 1.3*: The two behavioral interface specification formalisms lead to two different methods for verification of Creol models. In both methods, we use the specifications to simulate the behavior of an open environment. The first method is based on metalevel strategies in rewriting logic. In the second method, we extend the existing Creol interpreter such that it can execute the synchronous parallel composition of a component with a specification acting as the environment; this gives a basis for model-based testing of Creol components (Goal 1.3.1). Both testing methods are implemented in Maude (Goal 1.3.2).

We show the applicability of the approaches by designing and implementing concrete examples. The examples and the experimental results are presented in the included research papers. In Paper #1, the example is a Maude implementation of the dining philosophers. In Paper #2, we use the same testing framework, but introduce a more extensive example of a distributed system for resource sharing, where nodes in a network have an initial amount of local resources, and borrow from each other in order to perform required tasks. In the Papers #4 and #5, we consider the second method for verification. We use different variants of a broker example, where a broker acts as an intermediary between a client and several providers of some service. We also use a simpler example tailor-made to show the effect of using modulo AC rewriting to reduce resource consumption when testing in an asynchronous setting.

1.2 Outline

In the following chapters I briefly give some background to introduce the context for the work reported in the research papers in Part II. Chapter 2 introduces object-oriented principles and Creol, with a special focus on the active objects concurrency model for Creol. The tools and frameworks that we develop are executable in rewriting logic; therefore, rewriting logic and Maude are introduced in Chapter 3. Chapter 4 gives a brief introduction to software verification, and the use of formal methods for testing, especially model-based testing. Chapter 5 describes our approaches to testing and the implementation of the tools and frameworks. Chapter 6 introduces each of the research papers and their main contribution. Chapter 7 concludes the dissertation with an evaluation of the contribution of this dissertation towards the goals stated in this chapter.

Chapter 2

Object orientation and asynchronous communication

The first object-oriented language is Simula 67 [DMN68] developed by Ole-Johan Dahl and Kristen Nygård in the 60s. Simula later inspired Smalltalk [GR83], which became important in popularizing many of the notions first introduced with Simula. Another notable object-oriented language also inspired by Simula is C++ [Str86] developed by Bjarne Stroustrup as an extension to C. The most well-known and used object-oriented language today is probably Java [GJSB00]. The family of languages considered object-oriented is quite large. It includes, e.g., Modula-3 [Nel91]—a language that has attracted less interest from industry than from research communities but has influenced languages such as Java and $C^{\#}$. Python and Ruby are examples of general-purpose object-oriented languages among the most popular programming languages today.

In all problem solving, so also in programming, an analytic approach is useful. Especially when developing large and complex programs, the issue of breaking down the problem into smaller portions is important to enable the programmer to tackle each aspect in relative isolation. In computer science, this process is known as "separation of concerns" [Dij74]. The creators of Simula also point to the importance of decomposition for dealing with large problems and systems [DMN68].

Mechanisms that aid such separation of concerns are modularization of program code into procedures, into objects and classes, and information hiding through data abstraction. Data abstraction is a mechanism for hiding the (concrete) implementation details of data. Abstract data types (ADT) is one form of data abstraction used in most programming languages, also in object-oriented languages, but in the latter data abstraction is also realized through *objects*. The object-oriented paradigm is claimed to have a range of good properties, but most essential seem to be that the use of objects as a structuring mechanism

1. supports separation of concerns through *information hiding*, and
2. in addition promotes code reuse through *inheritance* and *dynamic creation*.

Information hiding in object-oriented languages is achieved through the mechanism of *encapsulation*. In a paper from 1989 [Nie89], Nierstrasz tries to clarify the term “object-oriented”. Since at the time it was used in many different ways, about different programming languages, what exactly was meant seemed unclear. He concludes that the fundamental object-oriented concept is encapsulation and that all object-oriented languages exploit this idea to various ends. A defining

characteristic of encapsulation is that it is a process that serves to separate the contractual interface of an abstraction and its implementation [Boo04]. An object is an entity that encapsulates data and the operations for manipulating this data. In our setting, these operations are known as the *methods* of an objects and are invoked using *method calls*. Thus in an object both the internal realization is hidden and protected from interference from the outside, and also the operations that are available for the outside environment are explicitly defined. By only allowing interaction with a component through its interface it is protected from accidental interference.

Encapsulation helps dividing complex programs into smaller manageable units by keeping internal details hidden, but also contributes to the second, good property mentioned above, namely *code reuse*. Encapsulation promotes code reuse since an external user of a class relates to the public interface and not the internal implementation. The only condition for substitutability of an object (class) with another is that the interfaces match, and no consideration of the internal implementation is necessary. This is especially useful in an open distributed setting, where components on one node may be upgraded or replaced and should still be able to interact with other nodes as long as the interface contract holds.

Inheritance is another powerful feature of the object-oriented paradigm, which also promotes reuse. This is done by organizing classes in a hierarchy where a subclass inherits methods from its parent. An object of the subclass will then by default contain the methods declared in its parent, and may further extend the behavior defined in its parent by declaring additional methods and fields, or it may even re-declare existing methods to *override* the parents' behavior. This is made possible through *virtual binding* (also known as *late binding* or *dynamic dispatch*). A method is virtually bound if the body corresponding to a method invocation is selected at run time, as is necessary if the class of the object is not statically known. The class of an object o might not be known at compile time, at run time; therefore, when a call to $o.m()$ is made, it must be decided which implementation of m to invoke. This is typically the case when a method implementation in a subclass overrides the implementation in one of its superclasses. Languages may allow only single inheritance or also multiple inheritance, i.e., a class may have more than one parent. It is unsettled how to virtually bind method invocations in a class hierarchy with multiple inheritance. Creol has multiple inheritance, and a dynamic binding strategy for Creol is suggested in [JO05].

Dynamic creation of objects also exemplifies code reuse. In class-based object-oriented languages, objects are generated at run time from the *class* specified in the program code. Objects are *dynamic entities* that may be instantiated with different initial values, thus sharing common code for operations, but with different internal values of its fields. Note that objects have explicit identities, which must be known in order for other objects to call its methods. A class is considered a description of the structure of the objects generated from the class.

In [Nie92] two reasons that object-orientation is a useful approach to development of open systems are mentioned. First, the mechanism of *encapsulation* promotes manageability of complex systems by making it possible to decompose systems into manageable pieces with well-defined functionality. Second, we want open systems to be adabtable and *code reuse* promotes adaptability, e.g., through extending component functionality by inheritance.

2.1 Creol—an asynchronous object-oriented language

Creol [Cre07, JOY06] is a high-level object-oriented modeling language for distributed systems, featuring active objects and asynchronous method calls. The motivation behind Creol is to create a high-level language that in a natural way combines object orientation and distributed concurrency. The operational semantics is defined in rewriting logic [Mes92] and is directly executable in the rewriting logic system Maude. This provides a platform for executable Creol models, using prototyping and different methods for analysis; as simulation, test, and model checking as described in this dissertation.

Creol resembles an object-oriented *programming* language (for an example of a small Creol program see Paper #3, or [JO07] for more examples), but the language abstracts from implementation details and lacks the low-level constructs that are necessary for application development. For intra-object computations, Creol uses a functional language of side effect free expressions. The main purpose of Creol is to analyze the communication aspects of concurrent objects, and our focus is on Creol as a high-level *modeling* language.

The communication model of Creol is based on exchanging messages *asynchronously*. In synchronous communication, the object sending a message (the caller) must wait until the called object (callee) has returned the answer to the call before the caller may continue its activity. When using asynchronous communication, the caller may continue its current activity and may receive the answer from the callee at some later point in time. This implies that the caller and callee can execute concurrently. (On a single processor computer, the execution can of course not be actually concurrent, but will be interleaved.) The caller can thus proceed with some activity and may avoid unnecessary waiting for a reply. Additional mechanisms are then needed to handle the reply since it is no longer determined when the reply will be processed by the caller. Creol uses future variables (or futures [BH77, Yon90]) that reference a future result of an asynchronous method call to handle method returns. For example, an asynchronous method call in Creol has the syntax: $t!x.m(\vec{e})$, where t is a fresh label that provides a locally unique reference to the call, x is an object expression, m is a method name, and $\vec{e}$ is a list of the actual in-parameters to the method. After the call, the caller proceeds. To fetch the return values from the call at a later point, the programmer would use the syntax: $t?(\vec{v})$, where $\vec{v}$ is a list of variables. If the reply to the call identified with the label t has arrived, the return values (corresponding to the return-parameters of the method m) are assigned to the variables in $\vec{v}$ and execution continues. If the reply has not arrived, process execution will block unless the programmer uses an explicit release point as explained below. Note that, even if Creol's communication model is based on asynchronous method invocation, *synchronous* blocking calls can easily be achieved by combining the statements above for issuing the call and immediately trying to fetch the reply. The synchronous call statement $x.m(\vec{e};\vec{v})$ is defined as $t!x.m(\vec{e}); t?(\vec{v})$, for a fresh label t, and in effect this blocks execution until the reply arrives, with exception of reentrant self calls.

2.1.1 Internal synchronization of Creol objects

A main purpose with Creol is for modeling distributed systems at a high level of abstraction. Given that asynchronous method calls are the basic mode of communication, this calls for high-level local control structures (or constructs for internal synchronization). Therefore, Creol has explicit suspen-

sion constructs for controlling the internal execution in an object. Conceptually, each Creol object encapsulates its own processor. A Creol object acts as a *monitor*; at most one method body (thread) is executing at each point in time per object, and *processor release points* [Hoa74, BH73] can be used to influence the internal control flow in objects, i.e., between processes inside a concurrent object. To avoid uncontrolled interference and to achieve explicit control, no preemption of processes is allowed, the responsibility of releasing the processor lies solely with the executing process, which means that Creol has a *cooperative scheduling* policy. Processor release points are a basic programming construct in Creol, and are defined with guard statements [Dij75]. A programmer might use an $await$ guard on the reply statement above: $await\ t?; t?(\vec{v})$. Execution may pass the guard only if the reply to the invocation with label t has arrived, otherwise execution will *suspend* and the processor is released.

At any time, a set of processes may be waiting to execute in the object. After suspension of the active process, any previously suspended and now waiting processes of the object compete on equal grounds for the free processor, and may be selected for execution. New incoming calls to a Creol object will also compete with suspended processes for the processor and will eventually result in new activity in the object. By using processor release points Creol objects can dynamically change between active and passive behavior, i.e., both act as a server, executing its "own" processes and as a client serving incoming calls. All Creol objects have a special *run* method, after object creation a Creol object will execute this method and thus become active. If the run method terminates, the object becomes passive. The object may also suspend its present activity to allow external calls to enter.

Creol combines the two different models of shared state communication of values inside objects and only explicit exchange of messages outside objects. The inter-object communication of Creol is *asymmetric* in the sense that there are linguistic means to *send* a message (call a method), but not to *accept* a message (receive a call): objects are always input-enabled, insofar as they are always willing to receive a message. On the receiver, or callee side of a method call therefore each object possesses an input "queue" in which incoming messages are waiting to be served by the object. There are no means in the language to enforce priority of incoming calls, so by default the choice of which method call in the input queue that enters the object next is *non-deterministic*.

In addition to guards on call labels Creol also supports Boolean guards. A statement $await$ b, where b is a Boolean expression over local and object variables, leads to release of the processor if b evaluates to true. There is also a construct for explicit and unconditional release of the processor, $release$. The mechanism of processor release points gives a programmer control of process suspension and a flexible mode of programming, but introduces more non-determinism in the model. Since incoming calls also are selected non-deterministically this gives a model with much non-determinism. For modeling it is an advantage to have the possibility to abstract away from the details of scheduling and leave this unspecified, either for simplifying the model for analysis, or to leave the decision open for later stages in development. But this raises a challenge for testing and verification of the models. In this dissertation, we have studied techniques for dealing with non-determinism of the latter kind, arising from interface communication.

2.1.2 Active objects

In the Creol model, the concept of active objects is central. Nierstrasz in [Nie87] also uses the concept of *active objects* as a core concept for introducing a notion of concurrency in object-oriented programming languages; the proposed language is Hybrid. Rather than viewing objects as encapsulations of data, only acting as servers, in Nierstrasz' model objects are potentially active and the message passing itself is a basis for synchronization. Message passing always entails transfer of control, or the creation of new activity. Though Hybrid is based on an idea of *active objects*, the concurrency model of Hybrid is more similar to the multithreading model of Java, than to Creol.

The active objects model of Creol are more related to the Actor concept first introduced in 1973 by Hewitt, Bishop, and Steiger [HBS73] and further developed by Gul Agha [Agh96, AMST97]. In this model of concurrent computation, the basic active units are *actors*, which may receive and send messages and execute local computations. Agha also advocates modularity in analyzing open distributed systems and the actor model is defined as a model of concurrency, and is said to extend the concept of objects to concurrent computation. The encapsulation of activity in an object is similar to Creol; both Creol objects and Actors encapsulate state and a thread of control, may execute in parallel, and exchange messages. The mode of communication is asynchronous, as in Creol, but it is difficult to model message exchange in the form of method calls structured as invocations with matching replies in the Actor model since there is no clear distinction between invocations and replies. Asynchronous *message passing* does not provide the structure and discipline inherent in *method calls*. In the Creol communication model the abstraction mechanism provided by object-oriented methods is central, combining asynchronous communication with the structuring mechanism provided by the method concept. Creol's active objects are therefore more closely similar to "objects" in the traditional sense of object-orientation than Actors, which are a more special construct. In Actor-based languages there is usually one thread of control per object, whereas in Creol there may be multiple control flows in an object. The enabledness of the threads is explicitly controlled by constructs in the Creol language. This allows the programmer to gain some control of the interleaving of internal execution.

The Reactive Objects Language, *Rebeca* [SMSdB04, Sir06] is, as Creol, a high-level language for modeling concurrent and distributed systems, but is based on the Actor model. Rebeca is object-oriented with self-contained objects, called *rebecs* (reactive objects) which communicate with asynchronous message passing. (In [SdBMS05] the model is extended with a rendezvous-like synchronous message passing.) A model is a set of rebecs, concurrently executing and interacting with each other. Rebeca extends the Actor language through using classes as object templates, and by introducing a concept of components. A component is a subset of the rebecs in a model. The introduction of components enables compositional verification methods for Rebeca models.

In Rebeca, there is no suspension primitives, and execution of methods—in Rebeca these are called *message servers*—is atomic. This distinguishes Rebeca from Creol where programmer control over process suspension gives more flexible, but also potentially more complex, models that may be harder to verify. Model checking is used for verification of behavior of Rebeca models, and temporal logic is used for property specifications. Temporal formulas are built from assertions of the local state of a rebec using Boolean connectives and temporal operators.

Verifying components in arbitrary environments is hard, since the reachable state space of a component in an open environment may be very large. In [SMSdB04] this is called the *environment*

problem. In the methods for verification in this dissertation, we use assume-guarantee reasoning to tackle this problem; our specifications assert properties of the system under some environment assumptions. When the specifications are used as a driver for environment behavior, a restricted environment is simulated.

In [SMSdB04] another solution to this problem, namely *compositional minimization*, is used. Instead of using assume-guarantee specifications, a reduced environment is modeled. In Rebeca, the model itself is decomposed into a component part and an environment part. This decomposition of the model simplifies the model considerably, since for the rebecs that act as environment only their message sending capability has to be modeled and not their state or behavior. Thus, the environment can be specified as a set of external messages. A range of further abstraction techniques are used to make the model finite and to alleviate the state-space explosion problem.

2.1.3 Active objects vs. multithreading

Creol's concurrency concept based on active objects can be contrasted with the one of *Java* or *C#* where the notion of concurrency is that of multithreading. A thread in Java is a lightweight process that executes independently of other threads [Lea99].

One argument for preferring an active objects concurrency model over multithreading is that verification of multithreaded programs is hard [ÁMdBdRS02, CKRW99] in addition the synchronization features of Java has been shown to be insecure and as a result safety in Java becomes a matter of coding conventions rather than of language design [BH99]. Multithreaded programming is difficult in itself; the programmer must resort to the low-level mechanism of locking to prevent unwanted interference due to the preemptive scheduling mechanism of threads. With numerous objects this is typically complex and might easily lead to errors or unpredicted behavior.

With active objects it is the objects that encapsulate processing activity, in multithreading it is the thread. The executing entities are the threads also in active objects concurrency. However, with active objects the activity is never allowed to cross the border of the object. When an object issues a method call, the calling thread continues executing and the code of the method being called is computed concurrently in a thread located in the callee object. In other words, no thread ever crosses the boundaries of an object. This means that the boundaries of an object are at the same time the boundaries of the threads and so, the objects are the units of concurrency. This is also crucial for compositionality, the objects harness the activities and can be considered as bearers of the activities. In this way, the notion of encapsulation fundamental for the concept of object-orientation is preserved and extended from data to control.

If a thread that holds the processor lock tries to access the result of a call—the value of a future variable—and the value of the future has been determined—the method calculating the result has terminated—the thread just obtains the value and continues. If the value is not yet determined the executing thread might give up its lock via a conditional release statement and allow other waiting threads to grab the lock to the object. Alternatively, it might hold on to the lock and actively wait for the result, the choice of behavior depends on the program code.

In contrast, *Java* for example uses "synchronous" message passing, where the calling thread inside one object blocks and control is transferred to the callee. The difference between active objects and multithreading is subtle, and, it could be argued, mostly conceptual, since active objects can be mimicked using multithreading. On the one hand, in Java we achieve active object behavior

and asynchronous method calls by spawning a new thread for each method call and let one thread handle the call and the other thread continue the activity in the objects. On the other hand, we get synchronous communication in Creol by immediately after a method call, trying to access the result without releasing the lock on the object. But even if the two models can mimic each other to some extent, the choice of making the active objects model primary in Creol is important in that it provides a modeler with more suitable abstractions for concurrent systems.

The difference between the models can be illustrated by how locks are handled. The implicit locking mechanism built into Java, which comes into effect when a code block is declared `synchronized` is *reentrant* [Lea99]; a thread trying to enter a synchronized code block will enter if the lock is free, or if the thread already possesses the lock. This allows a thread initiating from a synchronized method to call methods on other objects and later call back to the first method or to other synchronized methods in the same object without deadlocking.

In the active objects model used in Creol, a thread does not cross the boundaries of its object. The activity in the callee that results from a call is therefore unrelated to the calling thread. Hence, the combination of synchronous calls and callback in Creol leads to deadlock. If an object O_1 calls $O_2.m()$ synchronously and $m()$ in its turn does a call back to O_1 we will get deadlock. O_1 is waiting for the return of its call to $m()$ and therefore the callback from O_2 will never start executing in O_1. In Java, the second call will be allowed to enter the object since it is the same thread, in Creol it will block since the relation between the threads is not stored. The lock handling mechanism in Creol therefore is simpler than in Java. Locks in Creol are binary, a lock of an object is either taken or free and is always per object, but might lead to deadlock in cases where in Java deadlock would not occur.

Since Creol locks are in principle binary, every process competes on equal grounds for acquiring the lock. The combination of object monitors and strictly binary locks prevents recursive calls. To enable recursion in Creol, a process that holds the lock and waits for the result to a call, will give way to a process that wishes to acquire the lock if and only if this process is the one the active process is waiting for.

2.1.4 Coordination through behavioral interfaces

The notion of *behavioral interfaces* is a generic notion for semantically enriched interfaces, i.e., that also specify behavior. The intended behavior of the components is given by behavioral interface specifications which in addition to giving syntactical requirements to method signatures—formal parameters, return values, their number and types—also include semantic requirements. In Creol a behavioral interface consists of a set of method names with signatures and semantic constraints on the use of these methods.

In many object-oriented programming languages, objects are typed by classes. The type of the object is its class and the class hierarchy is also a subtype hierarchy. This is not so in Creol; here an object is an instance of a class, but is typed by one or more behavioral interfaces. An interface specifies the behavior that must be supported by any class that implements the interface. The hierarchies for code (i.e., classes) and for behavior (i.e., interfaces) are distinct. This gives a flexible model, a class may implement different interfaces corresponding to different viewpoints on, or aspects of, the objects behavior and the same behavior can be implemented by different classes. This separation of classes and types also gives the programmer more control over code reuse. For

example as interfaces are not inherited by subclasses, a subclass of a superclass implementing an interface does not by default inherit the behavior of the superclass unless it is explicitly specified in the interface of the subclass. Also since a behavioral interface reflects one viewpoint or a partial specification of an object this is a useful mechanism for dynamic modifications, new functionality can be added by adding interfaces. Creol supports *multiple* inheritance both for classes and for interfaces.

Another interface-based coordination mechanism in Creol is the use of *cointerfaces*. This mechanism allows to statically restrict access to a method provided in an interface to callers of a specific type (i.e., of the cointerface). Thus, the callee can safely call back methods declared in this cointerface and Creol features a keyword $caller$, similar to $this$, but typed by the cointerface. In Papers #1 and #2, information given by the cointerface specification of an object under test is used to simulate arbitrary environment behavior for the object.

Chapter 3

Rewriting logic and Maude

In the first two papers of this dissertation, we discuss verification of behavioral components where the components are specified in rewriting logic and the simulations are executed in Maude. In the later papers, we consider testing of Creol components. The formal semantics of Creol is defined in rewriting logic and the Creol interpreter is implemented in Maude, therefore I will introduce Maude here.

Maude is a system for executing rewriting logic specifications; it supports simulation, model checking and verification. It is very suitable for formal modeling of communicating systems, in particular distributed and asynchronous systems. The authors of [DMT00] discuss several advantages of the Maude approach: Using Maude as a tool gives a flexible approach in that it can be used at many levels, ranging from debugging and improving prototype models to model checking of the full model. Using formal methods from the start is an advantage, since then corrections can be made early. As Maude models are executable, a formal model also becomes an executable prototype. Moreover, executability implies that the model must be specified with a high degree of formal precision, which is very useful for revealing inconsistencies and finding bugs that might otherwise go unnoticed. In addition to supporting simulation by direct execution of models, Maude also supports "exhaustive execution strategies" where not only one possible run, but also all possible runs from a given initial state can be explored. We use this feature for verification of model components in our testing framework.

We use Maude in different ways. In the first two papers, the approach is to use metalevel rewriting in Maude; instead of letting the rewriting engine execute a specification directly, we apply *metalevel* rules to control the execution of the object level specification. This gives very good control of the execution and we may for example inspect the current state or configuration between each rewrite step, and thus we can record the *history* of the execution. Moreover, the metalevel rules that control the execution of the configuration can include predicates over to the recorded history. In this way, a rewrite step may be blocked and execution halted if the rewrite step would cause a violation of some specification given as a predicate over the history; in this way we can trace errors in the execution.

The use of metalevel strategies in Maude enables us both to *simulate* an unknown environment, and *test* components by blocking execution if a violation of the specification is about to occur. A further advantage of the approach is that it can be done *transparently*; the original specification of components does not have to be altered in any way, but can be plugged directly into the metalevel simulation and test framework.

In Papers #4 and #5, which deal with testing and verification of Creol components, we extend the implementation of a Maude interpreter for Creol to run simulations and tests. This gives tighter control of the execution and is more efficient since there is no need for metalevel calculations between rewrite steps. On the other hand, it presupposes access to the Creol interpreter, and implies some modification of the runtime platform itself. But similar to the metalevel approach, it can be done transparently as there is no need to change the component specification that is tested.

In the following we introduce rewriting logic [Mes92] and Maude [CDE+02]. Maude is both a high-level declarative language based on rewriting logic, and a system for equational and rewriting logic computation. Rewriting logic has equational logic as a sub-logic [Mes92]. The equational logic chosen for Maude is membership equational logic [CDE+02]. Maude has been influenced by OBJ [GWM+00] but has a richer underlying logic compared to OBJs order-sorted equational logic [GM87].

The atomic sentences of membership equational logic are equalities $t = t'$ and membership assertions of the form $t : s$, stating that a term t has sort s. Such a logic supports sorts, subsort relations, subsort polymorphic overloading of operators, and definition of partial functions with equationally defined domains [CDE+02].

A *rewrite theory* is a 4-tuple $\mathcal{R} = (\Omega, E, L, R)$, where (Ω, E) together constitute the equational theory. The equational signature Ω specifies the operators, sorts and kinds in the theory. E defines equations between terms (equalities and membership assertions). L is a set of labels, l, and R is a set of labeled rewrite rules on one of the following forms:

$$\begin{aligned} &rl[l] : t \longrightarrow t' \\ &crl[l] : t \longrightarrow t' \textit{ if condition}. \end{aligned}$$

The condition of the latter is an associative conjunction that may include equations (both ordinary equations and matching equations), membership assertions and also *rewrites*, which must hold for the main rule to apply. Computation in a rewrite theory is by rewriting logic deduction, where rewriting computation with the rules in R is intermixed with equational simplification using the axioms in E. It is assumed that the equational part E is Church-Rosser and terminating. The rewrite strategy adopted in Maude is first to apply equations on a term to reach a canonical form, and then to do a rewriting step with a rule in R. This implies that for a given rewriting theory $\mathcal{R}$, rewriting operates on equivalence classes of terms *modulo* the axioms E.

Maude has a built-in mechanism to declare certain kinds of equational axioms in a way that allows Maude to use these equations efficiently. This is done by declaring operators (i.e., function symbols) to have certain *equational attributes*. Supported attributes are *associativity*, *commutativity*, *idempotency*, and *id* <*Term*>. These are used to specify the algebraic properties of the operators, resp. the identity elements. We use this mechanism in Papers #4 and #5 to specify associative and commutative prefixes of specifications to simulate observational blur.

A state configuration in RL is typically a multiset of terms of given types, specified in (membership) equational logic; it can be seen as a snapshot of the dynamically evolving system. The rewrite rules capture this dynamic behavior and describe how a part of a configuration can evolve in one transition step. A rewrite rule $t \longrightarrow t'$ may be interpreted as a local state transition rule stating that if a fragment or part of a state configuration matches the pattern t it may evolve into the corresponding instance of t'. Since rewrite rules apply to fragments of a state configuration, if rewrite

rules may be applied to non-overlapping fragments of the configuration, the transitions may in principle be performed in parallel. Consequently, rewriting logic (RL) is a logic of *concurrent* state change, but without assuming any specific model of concurrency, which allows for modeling any mode of communication be it synchronous or asynchronous. A number of concurrency models have been successfully represented in RL [CDE+02, Mes92], including Petri nets, CCS, Actors, and Unity. Maude has also been successfully used in numerous cases for formal specification and analysis of active networks, communication- and security protocols [DMT00].

When modeling systems, configurations may include system components modeled by terms of the different types defined in the equational logic. As an example a RL *object* can be specified as a term $\langle O : C \mid a_1 : v_1, \ldots, a_n : v_n \rangle$, where O is the object's identifier, C is its class, the a_i's are the names of the object's attributes, and the v_i's are the corresponding values [CDE+02]. This object model has been used in the first two papers in this dissertation, but, since Maude is flexible, we are not bound to one specific object model. For example, in the interpreter for Creol that we use in later work, a more layered object model is used where attributes are not merely name-value pairs and objects are defined as terms on the form:

$$\langle\, O : C \mid \text{Att: } S,\ \text{Pr: } P,\ \text{PrQ: } W,\ \text{Dealloc: } LS,\ \text{Ev: } MM,\ \text{Lcnt: } N \,\rangle$$

Here, the variable O is the object's identifier and C is its class. S is a substitution list, i.e., a list of variable to value mappings that represents the valuation of the object attributes. P is the currently running process—remember that a Creol object acts as monitor and only one process is active at a time. A process is an instantiated method body, i.e., the method code and a substitution list with values for formal parameters. W is a multiset of processes waiting to execute. MM is a multiset of unprocessed messages to the object (invocations or replies). LS and N are multisets of call labels and a counter used to generate unique call labels respectively. The call label is a handle (also known as a *future variable*) that allows the object to identify the reply to a specific call when it arrives later. The multiset LS of call labels are labels that are no longer in use, and ready for garbage collection, while N is a counter used to assure uniqueness of new call labels.

3.1 Reflection and the Maude Metalevel

Rewriting logic is reflective in the sense that there is a finitely presented rewrite theory $\mathcal{U}$ that is *universal*; any finitely presented rewrite theory $\mathcal{R}$ (including $\mathcal{U}$ itself) can be represented in $\mathcal{U}$. Let C and C' be configurations and $\mathcal{R}$ be a set of rewrite rules. We write $\mathcal{R} \vdash C \rightarrow C'$ to express that C may be rewritten to C' in the rewrite theory $\mathcal{R}$. Informally, a configuration C and the set $\mathcal{R}$ of rewrite rules of a specification in RL may be uniformly represented by terms $\overline{C}$ and $\overline{\mathcal{R}}$ at the metalevel. Using this notation, we have the equivalence [Cla00]

$$\mathcal{R} \vdash \mathcal{C} \rightarrow \mathcal{C}' \Leftrightarrow \mathcal{U} \vdash \langle \overline{\mathcal{R}}, \overline{\mathcal{C}} \rangle \rightarrow \langle \overline{\mathcal{R}}, \overline{\mathcal{C}'} \rangle,$$

which states that if a term C in the rewrite theory $\mathcal{R}$ can be rewritten to a term C', then the meta-representation of C in $\mathcal{R}$, $\langle \overline{\mathcal{R}}, \overline{C} \rangle$ can be rewritten to the meta-representation of C' in $\mathcal{R}$, $\langle \overline{\mathcal{R}}, \overline{C'} \rangle$, in the universal rewrite theory $\mathcal{U}$, and vice versa. Maude includes facilities to meta-represent a rewrite theory $\mathcal{R}$ and to apply rules from $\mathcal{R}$ to the meta-representation of a term C by so-called *descent functions*.

Metalevel rewrite rules may be used to select which rule from $\mathcal{R}$ to apply to which subterm of C. This is done by defining an interpreter function that takes a finitely presented rewrite theory $\mathcal{R}$, a term C, and a deterministic strategy S as arguments. The use of reflection is essential to our approach in Papers #1 and #2, where the testing method is based on the execution of reflective specifications at the metalevel.

Metalevel rewrite rules may further be used to modify a configuration or the rule set of a rewrite theory. Hence, metalevel rewriting can be used as a wrapper around a rewrite theory $\mathcal{R}$ in order to abstractly mimic a more elaborate rewrite theory $\mathcal{R}'$ extending $\mathcal{R}$. Further details on the theory and the use of reflection in RL and Maude may be found in [Cla00, CDE+02, CM02].

Chapter 4

System verification

A main subject of this dissertation is how to attain reliable software. In particular, we investigate how to verify that a Creol model correctly represents some specified behavior. This chapter gives general background to techniques for verification and testing, with special regard to work relevant to our approach.

A note on terminology The words verify/verification and validate/validation have slightly different meanings depending on the discourse community. Among theoretically inclined computer scientists, there is a tendency to reserve "verification" for methods that involves formal proofs and "validation" as a more loose term, which may include testing and other techniques that can be used to make sure that a program or some code is correct. In contrast, in a more general software engineering context, "verification and validation" are often taken together to mean the whole process of checking that (1) a software system meets its specifications (verification); and (2) that it fulfils the purpose that was intended (validation). More succinctly stated: Validation ensures that "you built the right thing", while verification ensures that "you built it right". In this dissertation, I use verification in the more general sense, meaning the process of ensuring that a system meets its specifications. In this sense, testing is a verification technique. Sometimes however I wish to contrast testing, which is incomplete by nature, with other verification techniques, e.g., model checking, which may be complete. I will use the term "full verification" for the latter kind of techniques, and the term "formal verification" for the narrow sense of verification that presupposes full formal proofs. Note however, that in some of the included research papers, the terminology deviates. In Papers #1, #2, and #4 we use "validation" to mean "verification" in the general sense explained above. In Paper #3, the word "validation" has an XML-specific technical meaning and in Paper #5 we use "verification" in the sense explained above.

Static verification There are several techniques to ensure reliability of software; one distinction is between *static* and *dynamic* verification techniques. Static verification techniques include model checking and program analysis, aka *static analysis*. Static analysis is a common name for different techniques for analyzing programs without executing them. The analysis is static, i.e., takes place at compile time, it presupposes inspection of the code of the program, and the result is approximations to the set of values or behaviors that arise dynamically at run time during execution of the program. A traditional application for static analysis is compiler optimization, where computations that are known to be superfluous or redundant at compile time can be avoided; a more recent application of static analysis is to analysis of software to avoid unintended or insecure behavior [NNH99].

Static analysis employs formal methods; another approach to ensure program correctness based on formal methods is to use Hoare logic. Hoare logic [Hoa69] is an axiomatic formal system for reasoning about program correctness by allowing rigorous deduction of formal properties of programs. A specification of program behavior is on the form of a so-called Hoare triple: $\{p\}S\{q\}$, where S is the program code (list of statements), and p and q are assertions about program states. The triple $\{p\}S\{q\}$ expresses that if a program is in a state where p holds, and then S is executed, then q holds afterwards. For a thorough treatment of the use of Hoare logic reasoning for Creol programs, see [Dov09].

Model checking [CGP99, BK08] is a formal verification technique where properties of a system are verified by systematically inspecting all states of an executable model of the system. It is a static technique in the sense that it presupposes access to the code of a program in order to build a model for it. The inspection of the states in the execution is done programmatically, by using a tool to execute the model and check the validity of the specified properties. These properties are given as formulas in some variant of temporal logic: Linear Temporal Logic (LTL) [Pnu77], a logic based on a linear time perspective (each moment in time has a single successor moment), or Computation Tree Logic (CTL) [CE82], a logic based on a branching time view (each moment has many possible successor moments). The logics have different and incomparable expressiveness, and lead to quite different model checking algorithms. Since a basic principle of model checking is the exploration of *all* possible states for a system, a major challenge is to tackle a large number of states. In particular, the number of states of a system may grow exponentially in the number of variables in a program, and exponentially in the number of components in systems using parallel composition. This fact is known as the *state space explosion problem* of model checking. There are different techniques to handle this problem, one is *symbolic model checking* [BCM$^+$92], where sets of states and transitions are grouped together to reduce the number of states in the model. Another technique is *abstraction*, which amounts to leaving out implementation details from a system model and do model checking on a more abstract (and thus smaller) model. *Partial order reduction* [Val92, Pel93, God91] is a third technique where the idea is to reduce the number of possible orderings of parallel or interleaved actions that must be analyzed, by finding only those where the order of actions have influence on the properties one intends to check. This is similar to what we do when we relax test specifications up-to observational equivalence. The idea is that when the difference between two sequences of events is in principle unobservable, we will treat them as equivalent. This enables us to make the search for possible error configurations in the state space more efficient. A quite different way to tackle the problem of state explosion is to consider only parts of the model, either by using bounded model checking—a symbolic model checking technique that introduces an upper bound on the length of possible counterexamples searched for [BCCZ99]—or by using random exploration of models. These latter techniques could be said to make model checking more similar to testing by giving up exhaustiveness [Gau05] and by only selecting a restricted number of cases for consideration.

Dynamic verification *Dynamic* verification techniques are techniques that check the behavior of software dynamically by executing it; or in a less technical vocabulary, techniques for *testing*. Testing has the advantage that it can be done by executing the *actual* implementation directly, and not by reasoning about a model, but it is incomplete since we can never be sure to cover all possible behaviors of a system by executing it. This is especially relevant for distributed and concurrent

systems with a high degree of non-determinism, where the number of test cases tends to be very large or even infinite. In a much quoted sentence, Dijkstra points out that testing can only show the presence of bugs, not their absence [BR70]. This, however, does not have to lead to a negative or pessimistic view on testing as an activity inferior to formal verification or model checking. Hierons et al. in the introduction to [HBH08] give two reasons for that: First that actually there is work that, with respect to some well-defined assumptions, shows that formalised testing *can* establish correctness, and second that in many practical situations there might be no viable alternatives to testing, e.g., where a valid model cannot be constructed.

Formal verification In contrast to testing, *formal verification* can give certain proof about a property of a system, but, when it is based on reasoning about the model of the system under test and not the implementation itself, it must necessarily rely on the assumption that the model of the system is correct [Tre96b]. Even if one may *contrast* testing to formal verification, it should be noted that testing also may rely on formal methods and be characterized as formal in itself. Traditionally testing has been perceived as a non-formal activity by many theoreticians, since it is based on *executing* code, rather than on formal reasoning and moreover it has also been considered ad-hoc and error-prone [BJK+05, Tre99]. Partly because of this, one has seen the need for bringing formal methods to bear on the field of testing. More recently there is an increasing tendency to see testing and formal methods as complementary [BBC+02, Hoa96]. Formal methods and software testing have been claimed to be two of the most promising approaches to achieve techniques that assist in production of reliable software [H+09]. In addition, formal methods have proved valuable to render testing practice a more formal, systematic discipline (see [Gau95, Ber91]). Formal approaches to testing have gained momentum in recent years, as for instance witnessed by the trend towards model-based or specification-based testing [DJK+99, BJK+05], where an explicit formal model or specification of the behavior of the system is central.

4.1 Testing

In broad terms, testing can be described as "the activity of observing systems, experimenting on them and drawing conclusions about the behavior based on what we see" [Bru05]. The main points in this definition are that we *experiment* with the system, typically by executing the whole or part of the system, and that we *observe* the results and form an opinion from the observations.

The practice of software testing can be classified along different dimensions. One dimension is the level of abstraction at which testing takes place. A common model for software development, the V-model, distinguishes four levels: At the lowest level is unit testing, where individual units of source code are tested, often by the programmer during development. A unit is the smallest testable part of an application. At increasingly higher levels are: integration testing, testing the integration of separate modules; system testing, comparing system specifications and the actual system; and acceptance tests, testing the whole system prior to delivery or release to make sure that it satisfies the user requirements.

A further dimension is which aspects of the system one is interested in. It might for example be its functionality, i.e., whether the system behaves as specified, the performance of the system, its robustness, its resilience to stress, or its reliability and availability. The difference in interests naturally leads to different viewpoints on testing.

Another distinction is between black-box and white-box techniques for testing. In black-box testing no knowledge is assumed of the internal details of the system under test, whereas in white-box testing different degrees of knowledge about the internal details (i.e., code) of the system under test might be used by the tester. Given that functional testing is done by providing some system with input and comparing the resulting output with the expected output without regard to the internal workings of the system under test, *functional* and *black-box* are sometimes used as synonyms in the context of testing.

A definition of testing that is more specifically directed toward functional testing is that testing is "... a set of activities that aim at showing that actual and intended behaviors of a system differ, or at increasing confidence that they do not differ" [PP05b]. When doing functional testing one must compare the result of the test runs, the actual behavior, with the intended behavior. If the intended behavior is given as a precise specification, this could be described as *specification-based testing*, sometimes also called *conformance testing*; the terminology reflects the fact that what is checked is the possible conformance between specification and implementation.

It is well-known that testing contributes significantly to the overall cost of software development; studies have suggested that testing often forms more than 50% of the total development cost [H+09]. Moreover, as more sophisticated engineering methods for development of software lead to larger and more complex software systems, the task of keeping track with testing methodologies becomes a challenge. Testing is claimed to lack a systematic engineering methodology and adequate tool support [VCG+08], and it is claimed that the complexity of testing even tends to grow faster than the complexity of the systems being tested [Tre08]. *Model-based* testing (MBT) is an approach to testing that is considered promising for meeting these challenges.

In model-based testing, a formal model of the system under test is central. The formal model is used to drive the testing, as a source for generating test cases or also as an *oracle*, i.e., a system that can determine whether the observed output resulting from execution of the test cases is consistent with the specification. Benefits of the approach are its proper formal foundation, its scalability, and that it allows a high degree of automation of not only test execution but also of test case generation [VCG+08, PP05a, Tre08, BJK+05]. Our approach to testing Creol models is a kind of model-based testing and we will return to this issue in Subsection 4.1.4 below.

4.1.1 Simulation of test environments

In Papers #1 and #2 we show how an interface specification can be used to directly *simulate* an environment for testing. A related, and much applied, technique for unit testing is the use of *mock objects* [MFC01]. For example in programming paradigms as Extreme Programming (XP) [Bec99], test-driven development is central. The idea briefly is to write test cases first, and then iteratively develop code and test until the tests succeed. For this one needs methods for unit testing where units are tested in isolation. Since this is done during development, i.e., at a point in time where the surrounding systems are not implemented or deployed, it is necessary to simulate the environment. Mock objects are special objects that mimic real objects for testing, and can be used to emulate the environment of an object under test. *Dummy objects*, *fake objects* and *stubs* are related concepts, which also indicate objects that are used in place of real objects for testing. The terminology is unsettled, but see [Mes07] which gives a taxonomy of the concepts. The distinguishing characteristic for mock objects in his terminology is that they support *behavior* verification by containing

some kind of behavior specification in the form of program code. There exist many frameworks for testing with mock objects, implemented for Java and .Net, and also for Perl, Smalltalk, Ruby, C, and C++.

In our case, we want to simulate open environments displaying *arbitrary* non-deterministic behavior within the limits given by the specifications. We do not want to explicitly program the behavior of dummy objects to mimic the environment. In our approach, (see Paper #2 Section 5) the environment is set up solely by defining a set of object identifiers representing abstract objects that will interact with the object under test. The behavior of the abstract objects is generated by the simulation framework, which feeds the real object under test with input that seem to come from the abstract objects.

4.1.2 Formal methods and testing

One could regard testing as a practical and non-formal approach to acquiring reliable software, in contrast to the use of formal methods for the same purpose; our perspective is instead to see how formal methods contribute to testing.

The authors of [BT01] distinguish three main schools within testing based on formal methods: First, testing for labeled transition systems (LTS); second, testing based on Mealy machines [LY96] (also known as the FSM-approach); and finally, testing based on abstract data type theory (ADT) [Gau95, Ber91, LGA96]. The ADT approach focuses on testing static aspects of systems, such as data structures and their operations. In this direction, the article by Gaudel from 1995 has been very influential. One of its main topics is the problem of test case selection. It is shown how from algebraic specifications on can formally define the notion of an *exhaustive test set* (one capable of detecting all errors), and how the *success* of this set equals the satisfaction of the specification. For *practical testing* this exhaustive test set is not usable since it is large or infinite and therefore one must chose subsets for testing. How subsets are chosen reflects hypotheses about the program behavior, for example assumptions of uniformity in behavior for some sub-domain(s) of input values, or assumptions of regularity of behavior of functions relative to the size of argument terms. So a main point is that tests cannot be seen in isolation, but always with regard to the underlying hypotheses. In Gaudel's terminology this pair of a set of hypotheses and a set of tests is the "testing context". The article [BGM91] shows a practical application of the theory. For a more recent discussion on how formal methods for testing rely upon hypotheses on the system under test, see [Gau05]. Here it is also stressed that even if an implementation passes all the tests in the exhaustive test set it does not necessarily mean that it satisfies the specification, the relation between a system passing all the tests and actually being correct, depends also on the underlying hypotheses.

In [Gau95], the problem of bridging the gap between the results of the tests (execution) and the expected output prescribed by the specification is addressed. This is known as the *oracle problem*, it entails comparing the concrete test output from the program with the specified behavior, which typically is an abstraction of the concrete output.

The other two approaches (LTS and FSM testing) focus on the dynamic aspects of system behavior. It is generally assumed that ADT testing can be combined with either of the two first approaches [GJ98]. Our focus is on behavioral testing, i.e., on the dynamic aspects of systems, and we consider *reactive systems*. A reactive system can be defined as a (software or hardware) system with non-terminating behavior, which interacts with its environment through observable events

[BJK+05]. One can model reactive systems in different ways; finite state machines and labeled transition systems are probably the most used formalisms. Many varieties of FSMs exist, most well-known and influential are Mealy and Moore machines [Mea55, Moo56]. Mealy machines have been widely used to specify reactive systems, and much research has been done on conformance testing for such machines especially in the areas of communication protocols, embedded controllers and sequential circuits, for an overview see [LY96].

The idea of labeled transition systems was introduced by Keller in 1976 as a conceptual model for understanding parallel programs [Kel76]. A transition system is a structure consisting of states and transitions between them; by adding labels to the transitions we get a LTS. It can be seen as an abstract machine that can be used to model the behavior of a system. The states of the LTS model the system states, and the labeled transitions model the actions that a system can perform. Formally, this can be defined as follows:[1]

Definition 1. *A labeled transition system is a 4-tuple* $\langle Q, L, T, q_0 \rangle$ *where*

- Q *is a countable non-empty set of states;*
- L *is a countable set of labels;*
- $T \subseteq Q \times (L \cup \{\tau\}) \times Q$*, with* $\tau \notin L$ *is the* transition relation*;*
- $q_0 \in Q$ *is the initial state.*

The labels represent the observable actions of a system. τ is a special label that denotes internal, hence unobservable, actions. States are also assumed to be unobservable, so the observable behavior of the system is represented by sequences of labels, with the τ-actions abstracted.

Labeled transition systems may be used for modeling the behavior of processes and they serve as the semantic model for several formal specification languages, e.g., CCS[Mil80], CSP [Hoa85], and LOTOS [BB87]. The LTS model is a more general descriptive model than the FSM model, and in the research literature a larger set of conformance notions is applied to conformance testing based on LTSs than to testing based on FSMs, where trace equivalence is the predominant conformance relation [TPvB97]. The underlying model for our work is that of LTS. For a discussion of the relation between the two kinds of models in the context of testing, and for methods of transformation between LTS and FSM, see [TPvB97].

In our approach, a specification in the trace language (of Papers #4 and #5) *is* a model of the intended behavior of the component. An expression in the specification language represents a labeled transition system. Since the operational semantics of the language is defined by rules given as labeled transitions, each expression in the specification language represents a labeled transition system. To illustrate, the specification $\mathsf{rec}\ X\ .\ (a\ .\ b\ .\ X + a\ .\ c\ .\ \epsilon)$, where a, b, c are communication labels, represent the transition system illustrated by the process graph in Figure 4.1. It is capable of displaying behavior, or perform the *traces* $a\ .\ b\ .\ a\ .\ c$ or $a\ .\ b\ .\ a\ .\ b\ .\ a\ .\ c$, etc.[2]

Formal testing theory for LTSs was introduced by De Nicola and Hennessy in [DNH84]. The notion of testing here serves as the starting point for describing the semantics of reactive systems rather than as a practical notion meant to be used for actual testing of programs. Their

[1] This definition is standard and we quote it here from [Tre08].

[2] Note that in this example we do not distinguish between input and output labels.

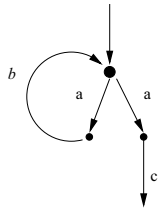


Figure 4.1: Process graph for rec X . $(a$. b . $X + a$. c . $\epsilon)$.

motivation was to find ways to describe a notion of observable behavior (for systems) through an *idealistic* notion of testing. Testing is used in the definition of testing equivalences/preorders, which define how to distinguish processes based on their *may* and or *must* properties. Even if the original focus was on describing semantics, many of their ideas and concepts are carried over to a practical application of testing, e.g., that of running a tester and observer in parallel, the influential distinction between *may* and *must* testing, and the concept of conformance described through implementation relations.

An *implementation relation* or *conformance relation* is a formal relation between a specification and (a model of) the implementation. Such relations are preorder relations, i.e., relations that are transitive and reflexive. A preorder relation that also is symmetric is an equivalence relation. Preorders can be interpreted as *implementation relations*. If for example S_1 and S_2 are in a preorder relation $\sqsubseteq$, such that $S_1 \sqsubseteq S_2$, then S_2 is an implementation of S_1 in the sense that S_2 is able to "perform the same actions upon its computational environment as" S_1 [Bru05]. An experiment e is a pair of input and the expected output $\langle i, o \rangle$, and a successful computation of the experiment is one where the actual resulting output from the computation (if it arrives) is equal to the expected output of the experiment. For example, $S_1 \sqsubseteq_{\text{MAY}} S_2$ means that for all experiments e, if S_1 *may* e (i.e., if the computation of (S_1, e) contains a successful computation) then also S_2 *may* e [Hen88].

By varying the content of the ideal notion of testing, more specifically by varying the ability of tests to distinguish processes, different preorders result. These distinguish processes in different ways. For a comprehensive exposition of different testing scenarios and the preorders each of these induce, see [Bru05] or [Gla01]. Preorder relations are key to conformance testing: A black box implementation and a formal specification are given, and one seeks to establish by testing whether the implementation correctly implements the specification. If it can be established that a preorder relation holds between the specification and the implementation then the implementation indeed implements the specification with respect to that preorder.

4.1.3 Observability and testing

The notion of *observability* is central in this dissertation. In general we assume that we do not have access to the internal details, i.e., the source code, of the components that we submit to testing. Since we take this black-box view on components, we only consider the behavior given by interactions at the *interface* of the component. Moreover, due to the asynchronicity of the communication model, the order of messages (interface events) may not be preserved during communication, as illustrated in Figure 4.2. This further means that the externally observed order of events does not

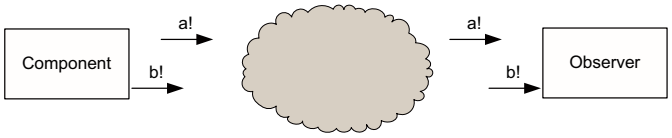


Figure 4.2: Observational blur.

necessarily reflect the order in which the messages were sent, therefore an observed "wrong" order of communication should not be taken to be an error. Note that the purpose is not to reconstruct some "correct" order of communication. When testing a component using the test setup described in Papers #4 and #5 and depicted in Figure 4.3, we control the communication. The test specification and framework play the role of both environment (generating input to the component) and observer (controlling output). But even if we have control over the communication in our test setup, we want to retain the external perspective. This is the reason that we allow reordering

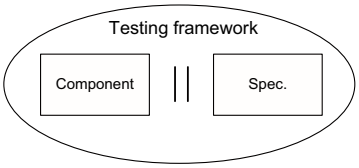


Figure 4.3:

of specified output events during testing. Technically, we achieve this by relaxing the specification up-to *observational equivalence*, and by testing output from an object up-to observability. The order of outgoing communication should not play a role for the test verdict since it is unobservable in principle, and the testing framework should take the fact that the order cannot be known externally into consideration.

The basic idea behind observability is quite simple: If one wants to understand what a program does one might start by observing it. From an epistemic point of view, only what is observable can be relevant, and this carries over to a computational point of view if the observer is a program or a program context. If two programs are observably equal, i.e., they have the same effect on any external observer, they are the same. It is what is observable from the "outside" that is relevant when we want to *use* the program.

The notion of observability, or observable behavior, is related to compositional verification techniques. The idea behind compositional verification was first formulated by Dijkstra in [Dij65]; essentially it is the analytical approach of decomposing a problem into its constituents. To verify a program, verify its specification by verifying the specification of its parts or subprograms and so on, until the parts are elementary components. But what Dijkstra also points out especially for program verification, and construction for that matter, is what he calls "the principle of non-interference".

This is the underlying assumption that correctness can be established by only taking the external specification of the parts into account, and not the internal details. Zwiers later formulated this idea as the principle of *compositional program verification*: "the specification of a program should be verified on the basis of the specification of its constituent components, without knowledge of the interior construction of those components" [Zwi89, p. 2]. This means that systems and parts of systems must only be specified in terms of observable behavior.

When formally reasoning about *processes*, where a process is understood as the behavior of some system, it is crucial that one is able to give a criterion for when two processes are equal. For example to be able to pass the verdict whether one process is an implementation of another. Such an equality criterion could be said to constitute the *semantics* of a process theory [Gla01]. Since in general, one might be interested in different aspects of processes one might employ a range of equality concepts for processes and accordingly there exists numerous semantics for processes. In the above-cited work, van Glabbeek gives a thorough classification and description of a range of different process semantics by defining different equivalences for processes. One way to motivate the different equivalences is through considering what behavior is observable under the assumption of a certain *testing scenario*; if we change the set of tests that may be applied, or equivalently the capability of the tester to observe what happens, we also change how processes can be distinguished. Some examples are

- Trace semantics/preorder. It is assumed that the process is a black box and we can only record the sequence of actions by recording each action as it happens, the display of the black box just shows the events as they happen.
- Completed trace semantics/preorder. Same as above, but it is also possible to distinguish completed and not (yet) completed traces.
- Observation semantics/preorders. In addition to observing the sequence of actions one can also take into account some of the intermediate states the system goes through.

Behavioral equivalence is a concept for comparing processes. A component of a program can be exchanged with another component with no change in the behavior of the program if the two components are operationally equal. This leads to a notion of equality that must respect this principle of extensionality: Two components are considered equal if it is impossible to distinguish them operationally, i.e., by executing them or interacting with them in some way.

Milner [Mil80] mentions observation and compositionality as the two fundamental ideas underlying his calculus of communicating systems (CCS). His aim is to give a purely extensional account of concurrent systems based only on what can be seen by an external observer. This leads to his notion of *observational equivalence*. It was first introduced in [HM80], where it is also argued that to compare programs it is not sufficient to consider their input/output relations, but that also intermediate steps in the computation must be considered since they can give different behavior in a larger context. This notion of equivalence has later been known as *bisimulation equivalence*; see also the extended version [HM85].

The *testing equivalence* of De Nicola and Hennessy [DNH84] is a coarser notion in that it distinguishes fewer processes. As mentioned above, the notion of test is fundamental; two processes are equivalent if they pass exactly the same set of tests. Different behavioral preorders on processes

are based on whether a process *must* pass a test, i.e., never fail, or *may* pass a test, i.e., both fail and succeed. These preorders induce the notion of testing equivalence.

The observational equivalence or bisimulation equivalence of Hennesy and Milner is very fine grained; it incorporates all distinctions that could reasonably be made by external observation. It can be argued that it is too fine to be applied in practice since it distinguishes processes that cannot really be differentiated by an observer [Bru05]. It induces a too complex testing scenario, where very powerful notions of testing are needed to determine whether an observational equivalence relation holds between two systems. Abramsky [Abr87] mentions in particular the notion of *global testing*, i.e., the ability for an experimenter to enumerate all possible runs of some test. Abramsky discusses a range of "reasonable testing notions", and shows how one might develop more powerful notions of testing to obtain a notion of equivalence based on *practicable* testing that coincides exactly with the observation equivalence of [HM80] and [HM85].

4.1.4 Model-based testing

If we take testing to be an activity to establish the correspondence (or the non-correspondence) between the intended behavior of a system and its actual behavior, it is clearly useful to have a precise notion of what the intended behavior of the system is. Often the intended behavior of a system is laid down by requirement specification documents, from which a model of the behavior can be deduced; in this sense any testing could be said to be *model-based* [Bin00]. In model-based testing, however, an explicit formal model of the behavior is central. Instead of "model" one could use the term "(behavior) specification", and *specification-based* is often used synonymous with model-based in this context.

Model-based testing is often described as a recent approach to testing (e.g., [BJK+05]), even if already in 1956 Moore published an article [Moo56] about black-box testing using finite state machines, where these machines are construed as models for real system or hardware devices, and where the idea is to test the models to learn about the systems. Another early approach to model-based testing, i.e., which aims at validating a program against a given formal specification using not only proof, but also testing can be found in [Ber91] (or [BGM91]). In the following we relate to a later, and more narrow understanding of *model-based testing* as it is described in several of the papers in [BJK+05] (see in particular [PL05]).

Our approach to testing in the two last papers of this dissertation could be characterized as model-based testing for Creol models, since Creol is a modeling language. To set our approach in context, I will in the following introduce MBT. The basic ingredients of MBT are

- The implementation under test (IUT), which is the real hardware or software system being tested. Note that the IUT is usually distinguished from the System under test (SUT), which is the IUT together with its *test context*, namely, elements one does not want to test, but which are needed to access the IUT [PL05]. The IUT is treated as a black box, and what is tested is the correctness of behavior on the interfaces.

- A specification of the behavior of the IUT. Or, in other words, a formal behavior *model*. This model is an abstraction or a simplification of a system under tests, and may include its environment [PL05].

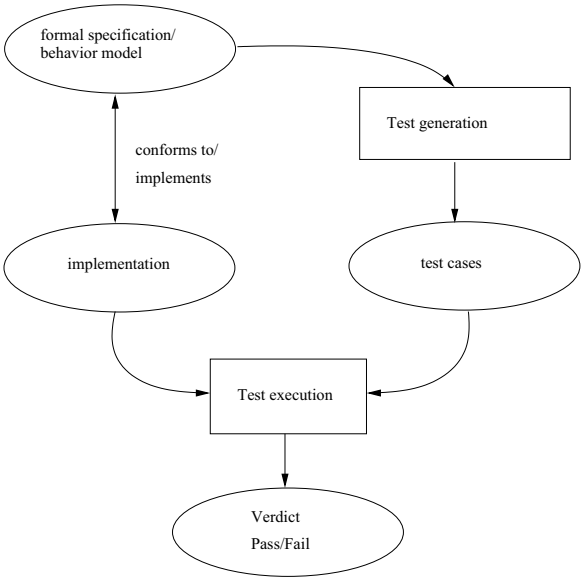


Figure 4.4: Schematic of model-based testing.

- In addition, there is the important assumption that there exists a formal model of the IUT; the real implementation is a physical object and is not amenable to formal reasoning. The assumption that any real implementation (IUT) can be modeled by a formal object is called the *test hypothesis* in [Tre99], and also *test assumption*. The idea originates from [Ber91] and [Gau95]. The assumption of the existence of a precise formal model of the system being tested is a basic underlying assumption for the approaches to testing that fall under the label "model-based". It is this assumption of a formal model of the implementation that allows formal reasoning about the relation between the specification and the implementation, and it is by this assumption that it is possible to reason about implementations as if they were formal objects. Given this assumption, conformance can be expressed as a formal relation between models of the implementation and the specification. Such relations are called *conformance relations* or *implementation relations*.

Figure 4.4 shows the relation of the parts involved in the model-based testing process.[3] There are three main steps of the testing process:

- *Generation of tests.* From a formal specification of behavior, test cases can be generated. A *test case* corresponds to one or many runs of the IUT, and is a structure of input and expected output behavior. A challenge for test case generation is to select "good" test cases, which means that they are cost efficient, i.e., cheap to derive, execute, and evaluate; and that they find the "serious" and "frequent" failures.

- *Execution of tests.* Executing a test case consists of giving as input to the IUT the input of

[3]The figure is standard and variants may be found in both [BJK+05] and [Tre08].

described by the test case generated from the behavior specification (model), and comparing the output of the IUT to the expected output prescribed by the specification. Based on observations of the outcome, a verdict may be passed. If the actual output corresponds to the expected responses the test is said to be successful, or in other words the implementation *passes* the test, otherwise the test in unsuccessful and the implementation *fails* the test.

A methodological point here is that to achieve a gain in efficiency by doing model-based testing, the model should be more abstract than the IUT itself, in the sense of being a simplification of the original system. If the model were not a simplification of the IUT, it would be no easier to check the model and use it for test case generation than to directly check the implementation. The test execution may therefore involve concretization of the input specified by the model before feeding it to the system, as well as abstraction of the resulting output to compare it with the specified output.

- *Assessment of conformance*. The goal of testing is to assess whether an implementation conforms, with respect to the conformance relation, to its specification, or model. Therefore a connection must be made between the outcome of tests (the pass/fail verdicts of the test cases) and the notion of conformance. A test suite that exactly distinguishes between all conforming and non-conforming implementations is said to be *complete*. This however is a theoretical notion unattainable in practice since completeness requires infinitely many test cases. A test suite that detects all non-conforming behavior is said to be *exhaustive*. A reasonable practical requirement is that a set of test cases should be *sound*, which means that all correct implementations, and possibly some incorrect ones, will pass the test cases.

The main virtue of model-based testing is claimed to be that it allows extensive test *automation*, not only for executing the tests, but more important for generating the test cases from the model. The goal of a test generation algorithm for model-based testing is to generate a test suite that is *provably* sound, and this is claimed to be the most beneficial aspect of model-based testing [Tre08]. In our work, we do not address systematic test generation algorithms. Concerning selection criteria for test cases, we make sure that we restrict the possible behavior to valid and well-formed traces, but have not evaluated selection criteria beyond that. Given that a trace is *well-formed*, we execute either one test (non-deterministically selected by Maude) or all (by doing a search). What we do therefore is either random selection, or the execution of all test cases. It has been argued that a random selection of test cases may in some situations be as effective as structural selection criteria [Pre05]. The testing tool TorX[BFdV+99] uses completely random test case selection. Executing all test cases that result from a specification may be possible in some cases, and a main point with our approach (see in particular Paper #5) is exactly to look for efficient methods for executing a large number of test cases.

Tretmans in [Tre08] describes model-based testing through the following characteristics: It is *formal, specification-based, active, black-box, functionality testing*. It is functionality *testing* in that it involves experiments (as opposed to formal reasoning) with the IUT, and the properties that are checked are *functional*, i.e., properties relating system responses to stimuli. It is *active* since the tester controls, and not only monitors, the activity of the IUT. The basis for testing is a *specification*, prescribing behavior, and the testing is *black-box*, only considering behavior at the interface of the component. That the testing is *formal* includes the fact that the specification is formal, that there is a formal definition of conformance, a well-defined algorithm for test generation and a correctness

proof that the generated tests are sound and exhaustive. Most of these characteristics apply to our approach to testing, with the exception that we have not addressed the issue of formally defining and proving a test generation algorithm, but have taken a more experimental or application oriented direction.

4.1.5 IOCO testing

We further exemplify the model-based approach to testing by looking at one specific application of it, namely input/output conformance (IOCO) testing, which is related to our approach to testing of Creol models. Ioco testing is a well-established approach for functional testing. It was introduced in [Tre96b] and a later and more thorough introduction is in [Tre08]. The general framework of the approach corresponds to the one described in the previous subsection. It is a *model-based* testing theory for labeled transition systems where LTSs are used as models for specifications, for implementations, and for tests. The new idea introduced by Tretmans in [Tre96a] and [Tre96b] to the theory of testing LTSs, is the one of explicitly distinguishing inputs and outputs of systems, where inputs are under control of the system and outputs under control of the environment. The motivation for doing this is to enable the construction of models with a closer link to reality since this distinction is natural for many real-life implementations.

Central to the ioco testing theory is the ioco conformance relation , it presupposes a model of interaction between a system and its environment with a distinction between actions initiated by the environment and actions initiated by the system, or in other words between input and output. This is exactly what we want; as we will test the behavior of the component and simulate the behavior of the environment, we need to distinguish actions according to whether the initiative or control is with the component or with the environment.

Given that the process model has this distinction, it is possible to define a formal conformance relation capturing the idea that for an implementation i to be correct with respect to a specification s, any output produced by i must have been foreseen in s, i.e., no output is allowed that has not been specified. Or to paraphrase Tretmans [Tre08]: an implementation i is ioco-conforming to a specification s if any experiment derived from s and executed on i leads to an output from i that is foreseen by s. Formally, this can be defined as follows:

Definition 2. *Let out$(\varphi$ after $t)$ represent the set of all possible output events that is specified by φ after execution of the trace t. Let out$(c$ after $t)$ represent the set of possible output events for the component c after execution of t. Let traces(φ) be the set of traces that the specification designates. Our conformance relation conf is defined as follows:*

$$c \textit{ conf } \varphi \Leftrightarrow_{def} \forall t \in \textit{traces}(\varphi) : \textit{ out}(c \textit{ after } t) \subseteq \textit{out}(\varphi \textit{ after } t)$$

The model for specifications used in [Tre08] is *labeled transition systems with inputs and outputs* which are normal labeled transitions systems with the additional property that input and output labels are disjoint. Implementations are modeled by *input-output transition systems* (IOTS). An IOTS is a variant of the input/output automata introduced by [LT98], and is a labeled transition system with inputs and outputs where in addition all input actions are enabled in any reachable state, i.e., inputs are never refused by the system.[4] Finally, a test case in this setting must be able

[4]This can be achieved by adding self-loops for all input labels to all states aka angelic completion, or by adding a

to do three things: provide input to the IUT, observe output from the IUT, and it must be able to observe quiescence if there is no output. A *quiescent* or *suspended* state is a state where the system cannot autonomously proceed. Accordingly, a test case can also be modeled by an IOTS, but with inputs and outputs exchanged and additional states for the verdicts pass and fail.

The ioco testing theory has been influential and several variants of the implementation relation exist. Tretmans in [Tre08] lists some, including ioco relations for real time systems and for hybrid systems, and in [FTW05] also relations for symbolic ioco testing. A number of test-tools are based on variants of the ioco testing theory, such as TGV [FJJV97], TestGen [HT99], and TorX [BFdV+99, TB03]. In symbolic ioco testing, the idea is to lift the family of testing relations to the level of *symbolic* transition systems in order to avoid state explosion during test generation. This approach could be further investigated for Creol testing.

4.2 Testing Creol models

In theory, model-based testing proceeds as described above, in practice the models are used differently. Compared to the description of the model-based testing process above, our approach described in Papers #4 and #5 deviates somewhat. First, we do not test an implementation of a system, but a model of a system. Our testing method targets model testing and not program code testing because Creol models are not plain models of behavior in the form of LTSs, but imperative underspecified models. The similarity of Creol and an object-oriented imperative programming language, together with Creol's expressiveness, allow for models that are structurally close to the systems being modeled but also lead to a need for testing these models as one would test programs.

In Creol, specific network behavior and particular scheduling of execution threads in objects are abstracted in the model. The asynchronous communication model and the under-specification of the scheduling of objects are sources of non-determinism in the executable model, and contributes to their complexity. Assume as given a Creol model that specifies a distributed system. We want to test the behavior of one component of the model in isolation. The non-deterministic behavior of the Creol model implies that the number of possible test cases that may be generated for a component is potentially very large. We have developed an executable framework for testing Creol components, where the number of test cases is restricted by the formal specification of the component. The formal specification language is in terms of observable interface behavior. Our approach specifically targets restriction of behavior by explicit scheduling of the order of incoming communication and validation of the corresponding output behavior. Thus, we can define precisely the scheduling of input and through this test internal synchronization properties of the object.

In the schematic presentation of model-based testing above, there seems to be a strict division between test case generation and test case execution. In our approach, we do not first generate the test cases from the specification and then later execute them for so passing a verdict. When test generation and test execution take place in two separate phases it is often described as *offline testing*, when both processes are merged into one phase it is called *online testing* (or "on the fly" testing). Online testing is a useful or even necessary method in the case of reactive systems when a large state space makes it infeasible to generate a test suite. Our method for testing Creol models is a kind of online testing.

special error state with transition to this error state for all input labels, or by introducing a special chaos state that all non-specified inputs leads to, and from which any behavior is possible aka demonic completion.

In light of what has been said in this chapter, our approach to verification of Creol models could be characterized as model-based testing for models where we test for conformance, in the sense of input output conformance, of a model component to its specification. Our approach shows how, in practice, ioco testing may be applied to distributed object-oriented systems with asynchronous method calls, exemplified by applying the method to Creol. Since Creol is a modeling language, it is not programs, but models, that are tested.

The authors of [AGSS08]—which also treats the subject of testing Creol models—point out that the ioco conformance definition presupposes synchronization of the IUT and the environment. This seems to be a bad match for our setting, using Creol, where we have underspecified models with a high degree of asynchronicity. But this synchronization is exactly what our testing framework adds to the Creol setting. Our specifications are a simple way to introduce synchronization constraints on components in order to allow testing for input/output conformance. See Subsection 5.2.1 for an implementation level discussion of the kind of scenarios that constitute a failed test case in our implemented test framework.

The approach to testing systems modeled in Creol taken in [AGSS08] on the other hand is based on the notion of *queued* testing. Input actions emitted from the environment are put in a queue and processed in any order determined by the implementation. A test verdict is reached by observing an interleaving of input and output events. The theoretical basis for queued testing was introduced in [PY02]; it is an expansion of ioco testing and shows how to deal with testing for systems with asynchronous I/O. For unsynchronized systems, the assumption is that a tester (or environment[5]) can never prevent a system under test from producing output, and the system should not block input from the tester. Therefore input from the tester and output from the system may occur simultaneously and must be queued in separate buffers between the tester and the system. In queued testing, the testing process is divided in two processes where one supplies inputs to the IUT from the input queue while another one is reading outputs from a queue. This separation results in a weaker conformance relation than synchronized ioco since relations between input and output cannot be captured. By expanding on the work in [PY02], Aichernig et al. show how by dropping the need to distinguish between input and output while monitoring events, it is possible to capture relations between input and output.

4.2.1 Related work

There are descriptions of related work in the included research papers in Part II; here I just mention a few additional publications of special relevance for testing of Creol models.

The related work [SAdB+08] pursues a similar goal of achieving more specific test scenarios for Creol, by investigating how different schedulings of object activity restrict the behavior of a Creol object. Their focus however is on how *intra*-object scheduling can lead to more specific test scenarios. Their test purposes are given as assertions on the *internal* state of the object, whereas ours are purely in terms of observable behavior.

In the articles [GAJS09] and [AGSS08] it is shown how different testing techniques can be employed to check for conformance between a Creol model and an industrial distributed system implemented in C. In the first article, the technique of *dynamic symbolic execution* is used to test for

[5]We have mainly used the term environment, since in our framework the tester plays the role of an artificial environment for the IUT, as well as being an observer of events emitted by the IUT. Tester is a more general term.

conformance between the Creol model and the implementation. Instead of restricting the behavior of the model, as we do by restricting the number of test cases by the formal specification, they use *dynamic symbolic execution* which seems to scale well. The authors point out that dynamic symbolic execution in general cannot deal with arbitrary non-deterministic interleavings of executions, but is applicable to Creol models since Creol provides the appropriate level of concurrency control.

In the second article, the authors use the same case study, and show how to instrument existing Creol models for testing. Aspect-C is used to insert event recording points into the existing code of the implementation under test. The model is likewise instrumented with synchronization points. A tester process is used to replay the recorded events in the model and it synchronizes the events of the model with the events recorded by the tester, only allowing the model to proceed beyond synchronization points if the corresponding event was recorded in the implementation under test. Thus, conformance of the implementation and the Creol model may be verified.

Two recent Ph.D. dissertations are related to our work. In the dissertation "Passive Testing with Parallel Object-Oriented Software Models" [Sch10], Rudolf Schlatte formulates theories and testing techniques for model-based testing of reactive systems. The modeling language used is Creol and the testing technique is *passive testing* or runtime verification. In passive testing one does not supply inputs to the SUT, but the system is observed during normal operation and the formal model is used as a test oracle only. This allows for testing that is minimally invasive so that existing Creol models can be used for testing without having to rewrite, or restructure them. In the methods developed in this dissertation there is also no need to rewrite the models for testing. But, in contrast, the approach in this dissertation is input-output testing, where a formal model is used both to generate input to the SUT and control output, which gives the possibility to experiment with different kinds of test input.

The dissertation "Testing Concurrent Objects" by Andreas Grüner [Grü10] (under preparation), considers testing of concurrent, asynchronously communicating objects, as in our work. The author introduces a test specification language for unit testing, the specification language allows to specify behavior (restricted to observable interface behavior) for units under test. In addition, the specification language can be used to automatically generate a test program that can test for a component's conformance to the specified behavior. This resembles our approach, but the design, and in particular the concurrency model, of the underlying language is quite different since it is not Creol, but a Java like language (Japl) which is a subset of Java.

Chapter 5

Solutions

The implemented solutions are described in the articles; we give some further details and comments here. The implementation of XML for Creol is described in detail in Paper #3, and will not be covered in this chapter.

5.1 A metalevel framework for simulation and testing

This framework is described in some detail in Papers #1 and #2. The following additional comments relates to the latter of the two. The implementation builds on [Axe04] and is a more developed version of the implementation described there. The main idea is to use abstract assume guarantee specifications given as predicates over the observable communication history of the components to simulate the behavior of an open environment. This environment is then used as a test bed for an actual component. The test framework is implemented in Maude by defining and combining execution strategies at the metalevel (see Section 3.1). A high-level representation of the architecture is given in Figure 5.1. In the figure, $\mathcal{R}_1$ is a set of object level rewrite rules for generating environment behavior, i.e., for generating arbitrary messages from the environment. These rules apply to the configuration $\mathcal{C}_1$, which consists of a term (`Envir`) that represents the environment. $\mathcal{R}_2$ is the set of object level rewrite rules applicable to the concrete objects in the configuration $\mathcal{C}_2$, which is the component under test. α_1 and α_2 are the interface alphabets, i.e., the sets of invocation and completion (return) events associated with the environment term, $\mathcal{C}_1$ and the component under test (CUT), $\mathcal{C}_2$, respectively.

P_1 is the predicate that (observationally) specifies the assumptions on the environment, and P_2 is the predicate that specifies the guarantees of the CUT. P_1 and P_2 are inferred from the interface specification of the component and are predicates over the history of the execution, restricted to the respective alphabets, as indicated in the figure by the projection h/α_n on the history.

Running a test then consists of metalevel execution combining several execution strategies.The metalevel strategy $\mathcal{S}_{restrict}$ restricts rule application from $\mathcal{R}_1$ to acceptable environment behavior, by only allowing behavior consistent with $P_1(h/\alpha_1)$. This provides an abstract, open environment which may behave in any way that does not violate the predicate P_1. The other strategy $\mathcal{S}_{test}$ will halt the execution and produce an error term if the predicate does not hold, this provides a way to test that the CUT does not violate P_2. To summarize (the numbers refer to Figure 5.1): The metalevel strategies 1) apply to the metalevel configuration in 4) and control application of the object level rules in 2). The rules in 2) apply to $\mathcal{C}_1$ and $\mathcal{C}_2$. Observable changes in the configuration

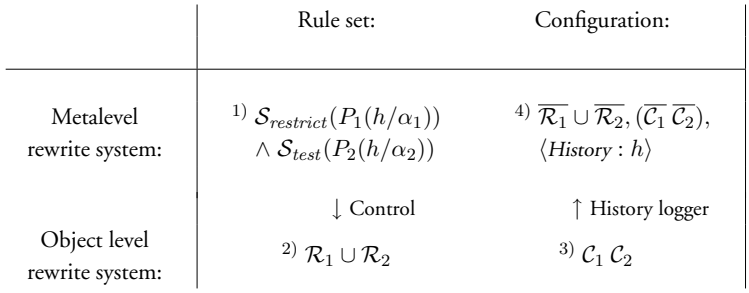


Figure 5.1: Reflective testing of observable behavior in open environments.

$\mathcal{C}_1\ \mathcal{C}_2$ are logged and added to the history object in the metalevel configuration in 4) which then again is part of the input to the metalevel execution strategies.

5.1.1 Implementation

At the Maude code level, the testing framework consists of several Maude modules where the core is the *metaengine* module, which contains metalevel rules for controlled execution of the object level component undergoing test, and for generation of environment behavior as described above. See Figure 5.4 on page 44. A test run is executed by calling a function `start` which takes as parameters

1. a reference to the configuration that is being tested (`MOD`), and to its initial state (`T`),
2. an upper bound (`I`) on the number of applications of the metaengine rule,
3. a predicate (`A`) describing the assumptions on the environment, and
4. a predicate (`P`) describing the guarantee of the component given the assumptions.

We give further explanations to the parameters in the following:

1: The configuration undergoing test consists of Maude code for the concrete object level CUT (in Paper #2, this is a `Node` as specified in Figure 2 of that paper), and a specification of the environment. The environment specification contains identifiers for abstract objects. These identifiers are used as values for senders in the generated invocation and completion (return) messages to the CUT. The environment specification further contains the object identity of the CUT and its interface alphabet to allow the generation of syntactically correct messages (method calls). A sample test configuration is given in Figure 5.2 (p. 41). Different method invocations and completions can be generated by modifying the environment specification `env` in this module. In the current version of the framework, the parameters of calls are randomly selected from a predefined list of values as given in the listed code, further development could include more sophisticated mechanisms for parameter generation.

2: When testing non-terminating applications a bound on the number of rewrites is useful since it allows to stop the execution after a certain number of steps and print the result configuration and a trace of the communication history.

3 and 4: The predicates describing assumptions and guarantees are specified using a special constant `H` as a placeholder. It is replaced by the actual communication history at runtime. The

```
mod TEST-CONFIG is

pr CONFIG-INIT .
pr OBJECT-RULES .

*** INIT init:Configuration contains the object(s) to test and the environment.
op init : -> Configuration .
op env : -> Env .

eq env =
  <E:Envir | absIDs: ('A1 ;; 'A2 ;; 'A3 ;; 'A4) , *** abstract Nodes
  sysIDs: (['N,((  *** concrete Node, (CUT) and its alphabet
  *** invocations from abstract objects (environment) to CUT
  meth('borrow , par(IntType, (int(1) # int(2) # int(3) # int(4) # int(5)) )) ;;
  meth('return , par(IntType, (int(1) # int(2) # int(3) # int(4) # int(5)) )) ;;
  meth('acquire, par(IntType, (int(10) # int(20)))) ;;
  meth('release, noPar));
  *** completions/returns from abstract objects (environment) to CUT
  (cometh('borrow , par(IntType, (int(1) # int(2) # int(3) # int(4) # int(5)))) ;;
   cometh('return , noPar) ;;
   cometh('acquire, noPar) ;;
   cometh('release, noPar)))] ),
  seed: seedVal > .

eq init =  < ob('N) :Node | local: initVal, brwd: noMap, state: free > env .

endm
```

Figure 5.2: Test configuration.

actual history is a `MsgList` that consists of of `invoc` and `comp` messages concatenated with an `@` operator. The code in Figure 5.3 shows how the assumptions and guarantees of Paper #2 are specified in Maude.

The metaengine execution rules in Figure 5.4 illustrate how, using the Maude metalevel, we can control in detail the execution of the object level module that is being tested. The function `start` sets up the initial metalevel configuration to which the main rule `exec` applies. In this configuration `MetaRep` is a wrapper for metalevel terms. It keeps track of which object levels rules are next to be applied, and of failed rules. `ObjectLevel` contains the object level configuration, which is useful to have available since the syntax of metalevel configurations is hard to read for an end user of the tool. `Hist` is the communication history, `Ctr` is a counter for rule applications. The terms `Success`, `Fail`, and `Done` indicate the status of the execution. Initially the status is `Success` and it remains so until either the chosen number of steps is reached or an error has occurred. If no errors occur within the chosen number of steps, the status is set to `Done`, and the resulting configuration is printed at the object level by applying the Maude `downTerm` function; this is handled by the `exec-done` rule (Figure 5.4).

The `exec` rule works as follows: First the result of the evaluation of `metaXapply([MOD], T, L, none, 0, unbounded, 0)` is bound to the term `RESULT`. `metaXapply` is a built-in Maude metalevel function which applies the rule with the label `L` to the (meta representation of the) term `T` in the module `MOD`. The result of an application of `metaXapply` is either `failure` or a 4-tuple containing the resulting term (see [CDE+05] for further details). In our case if the result is `failure` it means that the rule `L` could not be applied and we try the next rule, otherwise we check if the candidate rule for application `L` is a rule for environment behavior, i.e., one that should execute in $\mathcal{S}_{restrict}$ mode. If this is the case we update the metalevel configuration if the newly generated message does not conflict with the environment assumption, otherwise we try the next rule in the list. If the rule is an object level rule, it should execute in $\mathcal{S}_{test}$ mode, and we check if the guarantee predicate would be violated by applying the rule to the current configuration. If it would violate the predicate, we stop execution and report the error by setting the status to `Fail`. When the status is set to `Fail`, information about which rule caused the failure is also included in the status. In addition, the recorded history up to that point provides an error trace for the system run, describing how the specification was violated. If the guarantee predicate would not be violated by applying the rule, we apply it, update the configuration and continue execution.

As an example, a run of the system using the configuration and predicates as given above and restricting execution to 100 steps gives the result referred in Figure 5.6. In this particular case, the trace includes 7 messages, some generated by the environment, and some from the component under test, which is named `ob('N)`. Within the given number of steps, no violation of the guarantee predicate has been detected.

```
op H : -> History [ctor] .
op AccBeh : History -> BoolExp [ctor] .
op EnvAss : History -> BoolExp [ctor] .

var HIST : History .
*** The placeholder HIST is replaced the by the actual history, the MsgList ML.
eq CheckPredicate(AccBeh(HIST),ML) = AccBeh(ML) .
eq CheckPredicate(EnvAss(HIST),ML) = EnvAss(ML) .

*** Environment assumption.
op EnvAss : MsgList -> Bool  .
eq EnvAss(ML) = wellFormed(ML) and EnvA(ML) .

*** Definition of wellformedness of history
op wellFormed : MsgList -> Bool .
eq wellFormed(noMsg) = true .
*** invoc is always wellformed
eq wellFormed(ML @ (msg invoc MC from X to Y)) =  true .
*** comp: There must be one outstanding invoc msg to send a comp to .
eq wellFormed(ML @ (msg comp MC from X to Y)) =
  (len(PP((H / event(invoc,getMeth(MC)) / from(Y) / to(X)),ML)))
  > (len(PP((H / event(comp,getMeth(MC)) / from(X) / to(Y)),ML))) .

*** ASSUMPTION
op EnvA : MsgList -> Bool  .
eq EnvA(noMsg) = true .
eq EnvA(ML @ (msg invoc 'return(D) from Y to X)) =
  (( sum(PP( (H / event(invoc,'return) /  to(X) ),ML)) + (D asNat) )
  <= sum(PP( (H / event(comp,'borrow) / from(X) ) , ML ) )) .
eq EnvA(ML @ M) = true [owise] .

*** GUARANTEE
op AccBeh : MsgList -> Bool  .
eq AccBeh(noMsg) = true .
eq AccBeh(ML) = specPred(ML) .

*** The incremental guarantee part G' is expressed  by specPred
op specPred : MsgList  -> Bool .
*** Never lend out more than you had initially . (Guarantee)
eq specPred(ML @ (msg comp 'borrow(D) from X to Y) ) =
  (sum(PP( (H / event(comp, 'borrow) / from(X) ) , ML)) + (D asNat))
  <=
  (initVal + (sum (PP( (H / event(invoc,'return) / to(X) ),ML)))) .

*** Adhere to the assumption that you never return to someone
*** something you have not lent, from that specific Node !
*** I.e. the concrete object satisfies a stronger assumption than the environment.
eq specPred(ML @ (msg invoc 'return(D) from X to Y) ) =
  (( sum(PP( (H / event(invoc,'return) / from(X) / to(Y) ),ML)) + (D asNat) )
  <=
  sum(PP( (H / event(comp,'borrow) / from(Y) / to(X)) , ML ) )) .

eq specPred(ML) = true [owise] .
```

Figure 5.3: Specification of assumptions and guarantees.

```
op start : Qid Term Int Pred Pred -> MetaConfig .
eq start(MOD,T,I,A,P) =
  <MetaRep | curTm: T,curMod: MOD,labels: getRuleLabels(MOD),failRls: nil >
  <ObjectLevel | config: noConf >
  <Hist | h: noMsg,p: P,a: A >
  <Ctr | max: I,cnt: 0 >
  <Success> .

crl [exec] :
  <MetaRep | curTm: T,curMod: MOD,labels: L LABS,failRls: FR >
  <ObjectLevel | config: CFG >
  <Hist | h: ML,p: P,a: A >
  <Ctr | max: I,cnt: J >
  <Success>
  =>
  *** Check if rule may be applied,if not,continue with the next rule.
  if RESULT == failure then
    <MetaRep | curTm: T,curMod: MOD,labels: LABS L,failRls: FR L >
    <ObjectLevel | config: CFG >
    <Hist | h: ML,p: P,a: A >
    <Ctr | max: I,cnt: J + 1 > <Success>
  else
    if IsRestrictionRule(L) *** the rule L is a rule for environment behavior
      *** if the generated message does not break the environment assumptions
      if ((getNewMsgs(T,getTerm(RESULT),MOD,ML) == noMsg)
           or
         (CheckPredicate(A,ML @ getNewMsgs(T,getTerm(RESULT),MOD,ML)))) then
        *** then do the rewrite
        <MetaRep | curTm: getTerm(RESULT),curMod: MOD,labels: LABS L,failRls: nil >
        <ObjectLevel | config: downTerm(getTerm(RESULT),noConf) >
        <Hist | h: ML @ getNewMsgs(T,getTerm(RESULT),MOD,ML),p: P,a: A >
        <Ctr | max: I,cnt: J + 1 > <Success>
      else *** continue without altering the current term.
        <MetaRep | curTm: T,curMod: MOD,labels: LABS L,failRls: nil >
        <ObjectLevel | config: CFG >
        <Hist | h: ML,p: P,a: A >
        <Ctr | max: I,cnt: J + 1 > <Success>
      fi
    else
      *** The current rule is a rule for component behavior.
      *** Check if predicate would be violated.
      if ((getNewMsgs(T,getTerm(RESULT),MOD,ML) == noMsg)
          or
          (CheckPredicate(P,ML @ getNewMsgs(T,getTerm(RESULT),MOD,ML)))) then
        <MetaRep | curTm: getTerm(RESULT),curMod: MOD,labels: LABS L,failRls: nil >
        <ObjectLevel | config: downTerm(getTerm(RESULT),noConf) >
        <Hist | h: ML @ getNewMsgs(T,getTerm(RESULT),MOD,ML),p: P,a: A >
        <Ctr | max: I,cnt: J + 1 > <Success>
      else
        <MetaRep | curTm: T,curMod: MOD,labels: L LABS,failRls: FR >
        <ObjectLevel | config: CFG >
        <Hist | h: ML,p: P,a: A >
        <Ctr | max: I,cnt: J >
        <Fail | violatingRule: L,violatingMsg: getNewMsgs(T,getTerm(RESULT),MOD,ML) >
      fi  fi fi
  if RESULT := metaXapply([MOD],T,L,none,0,unbounded,0) / (J < I) .
```

Figure 5.4: Metaengine execution rules.

```
crl [exec-done] :
  <MetaRep | curTm: T,curMod: MOD,labels: L LABS,failRls: FR >
  <ObjectLevel | config: CFG >
  <Hist | h: ML,p: P,a: A >
  <Ctr | max: I,cnt: J > <Success>
  =>
  <MetaRep | curTm: T,curMod: MOD,labels: L LABS,failRls: FR >
  <Hist | h: ML,p: P,a: A >
  <Ctr | max: I,cnt: J >
  <Done | msgCnt: len(ML),resultConfig: downTerm(T,noConf) > if (J == I) .
```

Figure 5.5: Metaengine execution rules (contd.)

```
Maude> frew start('TEST-CONFIG, 'init.Configuration, 100, AccBeh(H), EnvAss(H)) .
frewrite in SIMULATION-TEST :
start('TEST-CONFIG, 'init.Configuration, 100, AccBeh(H), EnvAss(H)) .
rewrites: 9745 in 32ms cpu (33ms real) (295347 rewrites/second)
result MetaConfig:

<Ctr | max: 100,cnt: 100 >

<Done | msgCnt: 7,
        resultConfig:

<E:Envir |
  absIDs: 'A1 ;; 'A2 ;; 'A3 ;; 'A4,
  sysIDs: ['N,(
          meth('acquire, par(IntType, int(10) # int(20))) ;;
          meth('borrow, par(IntType, int(1) # int(2) # int(3) # int(4) # int(5))) ;;
          meth('release, noPar) ;;
          meth('return, par(IntType, int(1) # int(2) # int(3) # int(4) # int(5))) ;
          cometh('acquire, noPar) ;;
          cometh('borrow, par(IntType, int(1) # int(2) # int(3) # int(4) # int(5))) ;;
          cometh('release, noPar) ;;
          cometh('return, noPar))],
  seed: 1728214046 >

< ob('N) :Node |
  local: 5,
  brwd: (ob('A1),0) + ob('A4),0,
  state: free >
>

<Hist |
        h: (msg invoc 'borrow(int(2)) from ob('A1) to ob('N))
        @ (msg comp 'borrow(int(2)) from ob('N) to ob('A1))
        @ (msg invoc 'borrow(int(2)) from ob('A4) to ob('N))
        @ (msg comp 'borrow(int(2)) from ob('N) to ob('A4))
        @ (msg invoc 'release() from ob('A2) to ob('N))
        @ (msg comp 'release() from ob('N) to ob('A2))
        @ msg invoc 'return(int(4)) from ob('A1) to ob('N),
        p: AccBeh(H),
        a: EnvAss(H) >

<MetaRep |
        curTm: '__['<E:Envir`|`absIDs:_`,sysIDs:_`,seed:_>['_;;_[''A1.Qid,''A2
        ...
        ⋮
        ... ,
        curMod:  'TEST-CONFIG,
        labels:  'nextSeed-restrict 'compborrow 'borrow 'return2 'release
                 'genMsg-restrict 'return1 'acquire2 'acquire1 'noborrow,
        failRls: 'return1 'acquire2 'acquire1 'noborrow >
```

Figure 5.6: Sample execution.

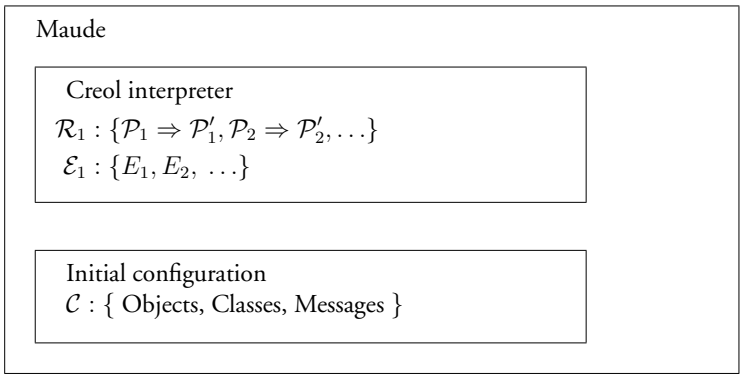


Figure 5.7: Standard execution of a Creol configuration in Maude.

5.2 A specification-driven interpreter for testing Creol objects

In this section we describe in some more detail the approach to testing taken in Papers #4 and #5. The Maude interpreter for Creol is a rewrite theory as described in Chapter 3. It consists of specifications of sorts and kinds as well as of operators over the sorts. This defines the terms of the theory. In addition it consists of a set of equations ($\mathcal{E}_1$) between terms and a set of rewrite rules ($\mathcal{R}_1$) for state transitions. A standard Maude state configuration ($\mathcal{C}$) representing a Creol model is a multiset of terms representing objects, classes, and messages. The Maude rewrite rules specified in the interpreter are of the form $\mathcal{P} \Rightarrow \mathcal{P}'$, (or on a conditional form $\mathcal{P} \Rightarrow \mathcal{P}'$ if cond), where $\mathcal{P}$ is a pattern that may match a part of the state configuration. Rewriting is intermixed with equational simplification, and thus rewrite rules transform terms modulo the equations of $\mathcal{E}$. The patterns of the rules in $\mathcal{R}_1$ match combinations of Objects, Classes, and Messages.

The standard way of executing a Creol model in Maude for simulating and observing behavior consists of giving as input to Maude: an initial configuration and the interpreter for the Creol language, and then let Maude rewrite this configuration. This allows for simulation of the model behavior. The model is executed and will, if it terminates, give as output a result configuration which may be inspected. For non terminating applications it is possible to set a limit to the number of rewrites. In addition one may use Maude's *search* command to search for specific result configurations (within a certain number of rewrite steps). The architecture for standard execution of Creol models in Maude is depicted in Figure 5.7.

The executable framework for *testing* Creol components that we have developed is depicted in Figure 5.8. It includes: the behavioral specification language formalized in rewriting logic and an extended version of the Creol interpreter that is specification-driven. For this specification-driven interpreter we introduce terms *Sp* for specifications and rules for evolving specifications ($\mathcal{R}_3$), corresponding to the ones of Tab. 9 in Paper #4. In $\mathcal{R}_2$, we add rules that match patterns on the form $(Sp \mid\mid O)\ \ \mathcal{P} \Rightarrow (Sp' \mid\mid O')\ \ \mathcal{P}'$ to test the object O with respect to *Sp* where $\mid\mid$ represents the synchronous parallel composition. Each rule evolves the state of a specification and the state of an object in a synchronized manner; an interface interaction of the object with the specification will only take place when it matches a complementary label in the specification.

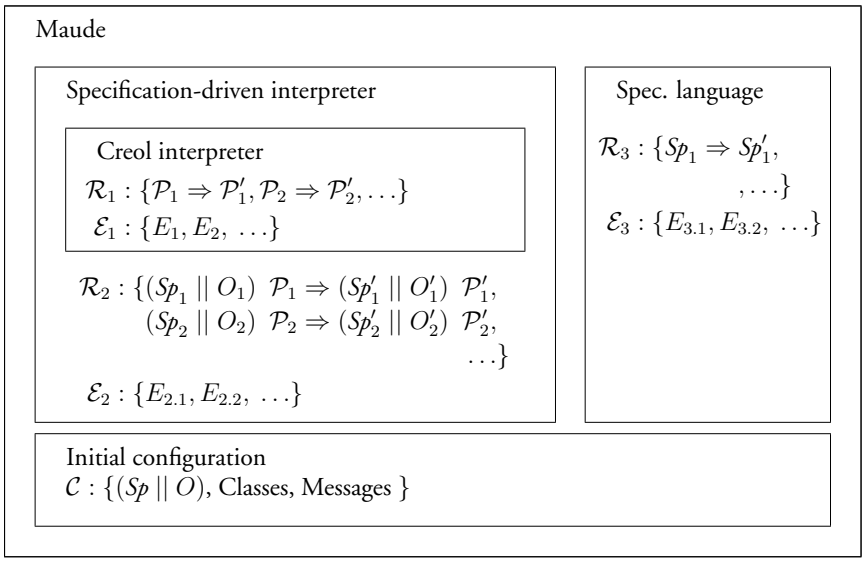


Figure 5.8: Specification-based testing of a Creol component.

For testing a component, instead of using the initial configuration as input to Maude, we extract one object `O` and its corresponding class definitions `Cl` from the model. This becomes the component under test. In addition we have a specification `Sp` of the behavior of the component. The CUT and the specification are then rewritten by Maude according to the rules of the modified interpreter. Input to the CUT is generated from the specification, and output is checked against the specification, internal activity is unmodified compared to the standard behavior. A test is executed by giving the Maude command:

```
rew ( Sp || O ) Cl . ,
```

where `O` is the term representing the Creol object. Maude rewrites the configuration, either resulting in an error reported when the component is about to execute an unspecified output. In that case the framework will return a trace of the interactions that lead to the error. Or, if no error occurs, execution will stop when no further rules apply. In the latter case, if the original specification is fully consumed this contributes evidence that the component conforms to the specification, in the sense that test execution of `O` only leads to output foreseen by the specification *Sp*. The conformance relation is input-output conformance (as described in Subsection 4.1.5).

Obviously even if no error is detected in one run this does not give certainty that the component conforms to the specification. Depending on the internal interleaving of the threads initiated by the method calls, different outcomes are possible. Maude's *search* command can be used to search, breadth first, for error configurations in the reachable state space. An error configuration is one containing an `errorMsg`, and the search is executed by giving the following Maude command:

```
search  in FRAMEWORK :  ( Sp || O ) Cl =>+
                        ( Sp' || O' ) Cfg errorMsg(S:String) .
```

By modifying the specification and using the methods of either rewriting the parallel composition of the CUT and a specification and observe the outcome, or by searching for specific states,

it is possible to test certain behavioral properties of specific objects by selecting specific objects and properties in a possibly large and complex object-based model. For instance, by altering the order of input labels in a specification we can check how different scheduling of input affects the execution of the object, see Paper #5 for a concrete example.

5.2.1 Implementation

For reference we include the Maude code for the specification-based interpreter, or Creol scheduler below in the Appendix A-1. In the following we comment on some selected issues, this section assumes some familiarity with the notation of the Papers #4 and #5.

Operational rules

The operational semantics for the synchronous parallel composition of an interface specification and a component is defined in Tab. 10 in Paper #4. The synchronization rule (PAR) of that table requires that, in order to proceed, the component and the specification must engage in corresponding steps. This is implemented by the following rewrite rules in Maude:

1. For local calls and returns:

   - `local-async-call`
   - `local-return`

   These are modified versions of the rules in the standard interpreter; the modified rules allow local calls and returns to happen without having to synchronize, in accordance with the PAR-INT rule.

2. For outgoing communication:

   - `PAR-remote-async-call` for outgoing remote calls
   - `PAR-return` for outgoing return

   In the rules for *outgoing communication*, we match the status of the object against a communication label in the specification, (using the function `matchCall` or `matchRet`. If we have a match, i.e., the communication label in the specification designates an event that might happen, the event is allowed to happen. The substitution returned by the match function is applied to the rest of the specification.

3. For incoming communication:

   - `PAR-incoming-call` incoming calls
   - `PAR-incoming-ret` incoming return

   In the rules for *incoming communication* we use the information in the communication label in the specification to generate a Creol message to the object. This is done by a call to the function `procLab` (process label) which returns a Creol message and a substitution (mapping the specification variables to the newly generated concrete values). The message and substitution are extracted by the functions `getM` and `getS`, respectively. The substitution is applied

to the rest of the specification. The substitution may be `noSubst`, which means that matching failed. Note that we generate both incoming *calls* and incoming *returns*. Matching of incoming calls trivially succeeds, but matching of incoming returns may fail if the corresponding outgoing call has not been seen at the interface earlier.

4. Error rules:
   - `PAR-ERROR` issues an error message if the specification requires input and the objects' next statement is a call out.
   - `PAR-ERROR-RETURN` issues an error message if the specification requires input and the objects' next statement is a return out.
5. Output prefix rules: In order to take the asynchronous nature of the communication model into consideration we test up to observational equivalence. Therefore in the implementation we wish to allow reordering of output events in specifications as specified by the rule Eq-Switch of Paper #4. We implement reordering of output events by defining *output prefixes* of specifications as associative and commutative using equational attributes in Maude. We need two extra rules for specifications with output prefixes.The following rules correspond to the rules for outgoing communication above:
   - `PAR-call-out-opf` for an outgoing call and a specification with an output prefix.
   - `PAR-return-opf` for an outgoing return and a specification with an output prefix.

In Section 5 of Paper #4 there is a more detailed explanation of two rules, namely `PAR-incoming-call` and `PAR-remote-async-call`.

Error-scenarios for test case execution

We here discuss the kind of scenarios that constitute errors in the parallel execution of a specification and a component. A specification defines the valid observable behavior under the assumption of a certain scheduling of input and there are basically three situations where a component may fail to conform to a specification.

1 A specification has at some stage been reduced to something like:

$$n_1\langle call\ c.m_1(x_1)\rangle?\ .\ (n_2\langle call\ e.m_2(v_1)\rangle!\ .\ n_3\langle call\ e.m_3(v_2)\rangle!)\ .\ \varphi$$

The meaning of this specification is that given that the CUT c receives the incoming call identified by n_1 it should afterwards do two outgoing calls (to methods m_2 and m_3 of the object e). It can in other words be seen as a synchronization constraint on the component. If in the course of execution of the component and specification, the component is in a state where it is about to issue a call out before it has processed the first incoming call, this is an error. The intuition is that the component has the wrong behavior if it actively does an action when it should rather wait for some other event. This first case is covered by the `PAR-ERROR` rule.

2 A specification has at some stage reduced to something like:

$$n_1\langle \mathit{call}\ c.m_1(x_1)\rangle? \,.\, (n_2\langle \mathit{return}()\rangle! \,.\, n_3\langle \mathit{return}()\rangle!) \,.\, \varphi$$

The specification states that after a call to m_1, two outgoing returns are expected. If the outgoing return takes place before the first incoming call happens, this is, as the previous case, considered an error. It could be argued that the CUT is not *responsible* for this error, since a method return from the component's point of view amounts to no more than that the computation is done and that a value is available to be read by the original caller which holds the (future) reference to the result. If the caller never decides to read the result, there is no outgoing interaction from the component. It might seem reasonable to say that when a method call is done at the callee side a message "passes" from the callee back to the caller's "message queue", and one might be lead to think that hence there is an outgoing interaction from the component. However whether the value resides on the side of the callee or in the "message queue" of the caller makes no difference. As long as the caller does not read the value it is not observed at the caller side.

Against this we argue that indeed the returns from the CUT are observable in the sense that we *may* construct or design an observer that reads (observes) the values, and our test specification is just such an observer. Thus we also consider "too early *returns*" to be an error on part of the component under test. One might also say that the component shows erroneous behavior when it makes the value available or lets the value be seen prematurely.

For both the above cases, however, the *order* of the outgoing communication events is *not* in principle observable, and therefore if the outgoing call identified with n_3 in the first example above is observed by the tester before n_2 this is not considered an error.

3 The third situation where a component is in error is also when it is in a state where it can do an outgoing call, or an outgoing return (i.e., conclude the computation of a call). The two previous items cover the case when the specification is expecting input at that point. But what if the specification specifies an output event that does not match the event produced by the component? The specification will then consist of an output prefix (or a choice that reduces to an output prefix) followed by either ϵ or followed by a specification starting with an input event $a?$ (or an input choice). If an output event that is ready in the component fails to match any of the output labels in the output prefix we have an error. One might say that this is no error since a component may later be able to do another output that matches the specification, but even so the non-matching output will still remain when the specification eventually reduces to an input label (or ϵ).

During the reduction of the output prefix an event that failed to match any of the output labels in the specification can not turn into a matching one. A new event that matches the specification will only contribute to make the specification more concrete. An event that did not match in the first place will not match later. So at the point where an input label is encountered in the specification the component will still want to do output and one of the above two error conditions applies. Since outgoing calls happen asynchronously, we effectively have an output queue for the CUT, and it is this output queue that synchronizes with the specification, and not the component. Thus a single object component will be able to proceed even if it wants to do a non-specified outgoing call. The call will "happen" in the sense that it is put into the output queue.

So even though we may report an error already at the point where a non-matching output event occurs, we do not do so, but let it be captured (eventually) by the two existing error rules. If no other output events match the ones in the specification, the specification will never be consumed and the erroneous output event will never be detected. Since testing is not *complete* and does not guarantee the absence of errors, this is acceptable. We do not require that all errors are detected. The example below illustrates this third kind of error scenario:

Example 3. *The* increment *component has the following specification*

$$\begin{aligned}\varphi_{INC} = \quad & n_1\langle call\ c.init(1)\rangle?\ .\\ & n_2\langle call\ c.incr()\rangle?\ .\\ & n_2\langle return(2)\rangle!\ .\\ & n_3\langle call\ c.incr()\rangle?\ .\\ & n_3\langle return(3)\rangle!\ .\\ \ldots\end{aligned}$$

If after the first call to c.incr() the component returns the value 7*, the testing framework will report an error, due to the* Par-Err-Ret *rule, but only if the correct output also happens.*

An implementation may enter a *quiescent* or suspended state which is a state from which it cannot autonomously proceed. This would be the case if the specification expects output from the component but the component is in a state unable to produce output. This can in principle only be observed by waiting infinitely long before concluding that the component will not produce any output, however for practical purposes this could be implemented using a time out mechanism. We have not done this in our testing framework.

Generation of input

An input label in the specification leads to the generation of messages to the component under test. This is handled by a function `procLab` (process label). It generates concrete values from the variables (or concrete values) in the specification label. The function builds a message, either an *invoc* message, i.e., a term representing a Creol method call, or a *comp* message, representing a method return.

For concrete specification labels (i.e., specifications without variables) the information in the specification language's *call* label is sufficient to generate a Creol `invoc` message for incoming calls. For incoming returns, the *return* label of the specification language contains a thread id and the return value. The thread id is a Creol label which contains the recipient of the return message, i.e., the original caller. There is no sender identity in a return label in the specification language, whereas Creol *comp* messages include the sender of the message. In the Creol operational semantics the rule for receiving a completion message discards the sender of the message anyway, so that information is superfluous.

If we consider specification labels with variables, at the stage where a *return* specification label is processed it will always contain a concrete value for the thread id. This follows from the assumption of well-formedness of the specifications and the definition of well-formedness stating that no value can be returned before a matching outgoing call has been seen in the interface. As soon as the outgoing call is seen the thread id variable in the specification will be substituted with a concrete

value. From this concrete label, we also get the receiver of the message, which is the object under test in case of only one object, however not when generalized to many objects.

For a *call* specification label with variables, we invent a caller and generate a thread id using this caller and a counter. The receiver is the object under test (in the case of one object), the method name is a constant and we generate arguments corresponding to the type of the method. As an example, we look at how the `procLab` function is defined for *invoc* messages:

```
ceq [gen-invoc-msg] :
    procLab(O , CT , call(Tid,R,M,Args) ? )
    =
    pRes( (invoc(Sender, Lab, M, DL) from Sender to Rcv ) , subst )
    if Rcv := getRcv(O,R)
       Lab := getInvocLabel(CT,Tid) /\
       Sender := caller(Lab) /\
       DL := getArgs(CT,Rcv,M,Args) /\
       subst := match((edPair(Tid,Lab) edPair(R,Rcv) zip(Args,DL)) , noSubst) .
```

An incoming call label: $Tid\langle call\ R.M(Args)\rangle?$ in the specification language is implemented by the Maude term `call(Tid,R,M,Args) ?`, which represents a call with thread name `Tid`, to the method `M` of object `R` with parameters `Args`. The `gen-invoc-msg` equation constructs this term by building the term `invoc(Sender, Lab, M, DL) from Sender to Rcv`, where the constituents are as follows: The receiver `Rcv` of an invoc message is always the CUT. `Lab` is the caller label on Creol format; it contains the object identifier of the sender and a counter to ensure freshness of new labels. We use a specific object with the object identifier `ob("EnvObj")` as the sender for generated messages. Finally the method parameters `DL` is generated by a call to `getArgs`. This function generates parameters to a method call if the specified call label contains variables or just the values if the specified call label contains values. To generate type-correct random parameters we could look up the signature of the called method from the class definition and generate random values within certain specified ranges. This function could also be tuned for optimization of test coverage. We have not addressed these issues in this version of the testing framework.

Implementation of the specification language

For further details of the implementation of the specification language as it is defined in Tables 8 and 9 of Paper #4 see the included code in the appendix, we here comment on some specific issues.

Distinction of output and input We distinguish between input and output interactions in the specifications. In the Maude implementation this is done by using different *sorts* for incoming and outgoing communication labels, and correspondingly for specifications:

```
sorts InLab OutLab CommLab .
subsorts InLab OutLab < CommLab .

sorts In Out Spec .
subsorts In Out < Spec .
```

Output prefixes We implement reordering of output events by defining *output prefixes* of specifications as associative and commutative using equational attributes in Maude. We introduce the following additional sorts:

```
sort OutPrefix .
subsort OutLab < OutPrefix .
```

and instead of the standard definition of the prefix operator . which is given by:

```
op _._ : InLab Spec -> In [ctor prec 45 gather(e E)] . (1)
op _._ : OutLab Spec -> Out [ctor prec 45 gather (e E) ] . (2)
```

we replace (2) with the following:

```
op _._ : OutPrefix In -> Out [ctor prec 45 gather(e E) ] . (2.1)
op _._ : OutPrefix Out -> Out [ctor prec 45 gather(e E) ] . (2.2)
op _._ : OutPrefix OutPrefix -> OutPrefix [ctor assoc comm prec 45 ] . (2.3)
```

Since `_._` is a prefix operation, we use `gather(e E)` to make it right associative thus avoiding ambiguity. Definition 2.1 above combines an `OutPrefix` with an `In` specification to yield an `Out` specification. Definition 2.3 is used to build output prefixes from outgoing communication labels (i.e., `OutLab`). Definition 2.2 is needed to handle cases where Definition 2.3 does not apply, i.e., where

$$\begin{array}{lll} \varphi_1 = & \ldots a!.X & \text{since the specification variable } X \text{ is of sort } \texttt{Spec} \\ & & \text{and not of sort } \texttt{CommLabel} \\ \varphi_2 = & \ldots a!.(sp + sp') & \text{since the choice } (sp + sp') \text{ is of sort } \texttt{Spec} \\ & & (2.2) \text{ also covers the case:} \\ \varphi_3 = & \ldots a!.epsOut. & \end{array}$$

note that Definition 2.3 is associative and commutative by the attributes `assoc` and `comm`.

With these definitions a specification with a commutative (reordering) output prefix can be given. (For technical reasons parentheses are used to syntactically distinguish output prefixes.) For example, the specification (for `a,b,c` of sort `CommLab`)

```
(a ! . b !)  . c ? . eps .
```

may lead to the behavior `a ! b ! c ?` or `b ! a ! c ?`.

Recursive specifications In Tab. 8 of Paper #4, observational equivalence of a recursive specification is defined by:

$$\mathsf{rec}\ X.\varphi \equiv_{obs} \varphi[\mathsf{rec}\ X.\varphi/X] \qquad \textsc{Eq-Rec}$$

Translated into Maude, this becomes the reduction rule:

```
rl[Eq-Rec1]   rec(xV,sp) =>  sp[rec(xV,sp) / xV] .
```

With this rule, however, the term `rec(xV , sp)` would rewrite as follows:

$$\texttt{rec(xV , sp)} \longrightarrow \texttt{sp[rec(xV,sp) / xV]} \longrightarrow \texttt{sp[sp[rec(xV,sp) / xV] / xV]}\ \text{etc.}$$

The inner `rec` expression will always be rewritten *ad infinitum*. To avoid this a *context* for specifications is introduced:

```
op <_> : Spec -> SpecContext .
```

By changing the `[Eq-Rec1]` rule to

```
rl [Eq-Rec]  < rec(xV,sp) > => < sp{[rec(xV,sp) / xV]}  > . ,
```

only the outermost `rec`-expression matches the rule.[1] (The notation `{[rec(xV,sp) / xV]}` indicates a set that contains one substitution.) This gives a *lazy* strategy for rewriting recursive expressions, meaning that in the specification:

$$\varphi \ldots \mathsf{rec}\ X.\varphi',$$

we only reduce $\mathsf{rec}\ X.\varphi'$ (by the rule above) when it is encountered in the course of executing the specification; then it is expanded to a substitution by the rule `Eq-Rec`.

The actual substitution of a recursive expression for a variable is likewise not effectuated until the variable is encountered during the reduction of the specification. Since a $\mathsf{rec}\ X.\varphi$ expression is of type `Spec` and we only have prefixing for communication labels, a `rec` expression is never followed by communication labels. This means that once a recursive step is taken, there is no need to return. A `rec` expression is always at the end of a specification, either as one of many choices in a choice expression, or alone:

$$\begin{aligned}\varphi_1 = &\quad \ldots \mathsf{rec}\ X.\varphi'_X \mathsf{or}\\ \varphi_2 = &\quad \ldots (\mathsf{rec}\ X.\varphi'_X + \varphi'')\mathsf{or}\\ \varphi_3 = &\quad \ldots (\mathsf{rec}\ X.(\ldots \mathsf{rec}\ Y.\varphi'_{X,Y}))\end{aligned}$$

(The notation ϕ_A indicates a specification that *may* contain the variables in the set A.) Recursion may nest as in the third case. The third case gives rise to the generalization of the rule `Eq-Rec`; a `rec` expression may have a set of substitutions (indicated by `{spSS}`).

```
rl [Eq-Rec-Sub] : < rec(xV,sp){ spSS } >(0,CT) =>
                  < sp{ [rec(xV,sp) / xV] spSS } >(0,CT) .
```

For each `rec` expression encountered during execution of the specification, a substitution is recorded. It is assumed that all specification variables are distinct, and that all are bound by a `rec` operator. By the rule `Eq-Rec-Sub` we add the new substitution to the set of substitutions. When a specification variable is seen at the start of the specification the substitution is applied:

```
rl [subst-var]  < xV{[sp / xV] spSS} >(0,CT) => < sp{ spSS } >(0,CT) .
```

Since all specification variables are distinct and bound by `rec` expressions, there is a unique value for `xV` in the set of substitutions at this point. Further we do not need to store the value for `xV` after the reduction step. Since the only use for specification variables is in recursions we know that reduction of the term `sp` in the rule above will not reintroduce the binding for `xV`.

[1] This technique was suggested by Marcel Kyas.

Chapter 6

Overview of the research papers

In this chapter, I summarize each of the research papers included in Part II. The papers appear as they were published originally. The work collected here can be organized in three parts, which, in chronological order, are

1. Verification of object-oriented components, using abstract interface specifications and meta-level rewriting techniques.
2. Data-based interface specifications for Creol components based on XML.
3. Verification of Creol components, using a concrete trace-based specification language and extending the Creol interpreter.

6.1 Part 1: Verification using abstract interface specifications

The first part is covered by Papers #1 and #2, which address the problem of verification of software components in open distributed environments. In particular, we consider how to overcome the difficulty of predicting and verifying behavior in unknown and possibly evolving environments.

6.1.1 Paper #1

Title: Validating behavioral component interfaces in rewriting logic
Authors: Einar Broch Johnsen, Olaf Owe and Arild B. Torjusen
Publication: In *Proceedings IPM International Workshop on Foundations of Software Engineering (FSEN 2005)*, volume 159 of *Electronic Notes in Theoretical Computer Science*, Elsevier, May 2006. [JOT06].

Summary
The components we consider here are modeled as distributed objects, which exchange messages asynchronously, using method calls. They have abstract interface specifications given as first-order logic predicates over the observable communication history of the components. We use these interface specifications to simulate the behavior of an open environment, where the simulation allows for arbitrary environment behavior within the bounds of the assumption on observable behavior.

The main idea is to use these underspecified formal descriptions of the components to simulate an open environment in which the components can be tested.

An object is typed by an interface, which specifies the methods it supports. An interface I can also require that a caller supports a specific type, a so-called *cointerface*, thus the cointerface is a static restriction on the objects that may call the methods of I. In addition, the interface has a predicate specifying the requirements on the communication history of the object supporting the interface. We show how such a predicate can be split into an assumption and a guarantee part with a strict distinction between assumptions that are the responsibility of the environment, and guarantees that are the responsibility of the object.

From the interface an *alphabet* for an object o can be deduced, which contains invocations to o of the methods specified in I and corresponding completions, as well as invocations from o to methods declared in the cointerface, and corresponding completions. We achieve a testing framework in Maude by simulating an open environment using the assumption part of the interface specifications to generate arbitrary environment behavior, i.e., input to the component under test. The input is generated from the alphabet that is deduced from the interface specification. We use the guarantee part of the specification to check the output from the components under test, and halt the execution if an error occurs.

The components are implemented in rewriting logic and the simulation is executed on the rewriting logic system Maude. We use *metalevel strategies* in rewriting logic to generate an environment for a component based on the specification. This environment is used both for simulating a run of the component and for testing the components. By using the Maude metalevel, we can monitor the behavior of the implementation *transparently*, without modifying the implementation.

The paper contains the following contributions:

- We show how to extend Maude models for object-oriented components with a notion of observable behavior by giving a formalism for behavioral interfaces for such components.
- We show how to simulate the arbitrary behavior of open environments within the bounds of the given specification by generating input to a system directly from the specification.
- We show how, by combining metalevel rewriting strategies for generating input and for monitoring the output, we can test that a system fulfills the guarantees in the specification, provided that the assumptions of the specification holds.
- This testing framework is implemented and we give an example with experimental results.

6.1.2 Paper #2

Title: Validating behavioral component interfaces in rewriting logic
Authors: Einar Broch Johnsen, Olaf Owe, and Arild B. Torjusen
Publication: In *Fundamenta Informaticae*, 82(4), 2008 [JOT08]

Summary

This is a revised version of the previous paper where the testing process is illustrated by a more extensive example of a distributed system for resource sharing. There is some overlap of this and

the previous paper. Even so, this paper is also included since my contribution has mainly been the case study and the implementation.

6.2 Part 2 : Data-based interface specifications for Creol components based on XML.

This part of the work is summarized in Paper #3 where we propose an extension to the Creol language for handling XML documents. This enables data-based interface specifications for Creol components and it lays directions for further work that must be covered for integrating XML in Creol. One particularly interesting lead to follow in this respect is to enhance the Creol language with *regular expression types*[HVP05], thereby allowing for the introduction of types for XML fragments as first class types in Creol.

6.2.1 Paper #3

Title: Towards integration of XML in the Creol object-oriented language
Authors: Arild Torjusen, Olaf Owe, and Gerardo Schneider
Publication: Published as Research Report 365, Dept. of Informatics, Univ. of Oslo, October 2007 (revised February 2008) [TOS07a]. The work reported here was presented as a poster at the annual Norwegian informatics conference 2007. A short version has been published in the proceedings: *Norsk Informatikkkonferanse (NIK 2007) 19–21 Nov, Oslo, Norway*, Tapir Akademisk forlag 2007. [TOS07b]

Summary
In this paper we propose an extension to Creol for handling XML (eXtensible Markup Language) [W3C00] documents. XML *documents* are ordered labeled tree structures containing *markup* symbols describing their content. The document structure is described by a document type or *schema*, which specifies a grammar for the document. Our approach is to introduce sorts for XML documents and schema in the functional sublanguage of Creol and define a *validate* function, within the existing type system.

The data model defined in XPath 1.0 [W3C99] is the basis for canonical XML, which we take as a starting point. We extend the operational semantics of Creol by extending the functional sublanguage in Maude with sorts and constructors for XML data. For specification of XML document types (i.e., grammars) there exist several *schema* languages. We implement the DTD [W3C00] language in Maude by introducing regular expressions to describe document structures.

We further present an algorithm for validating XML documents against XML schema; this enables Creol applications to use XML documents as a format for data storage and exchange. XML documents can be validated using the defined functions over the XML document type, but cannot be type checked through the Creol type system. Therefore, this is a first step towards integration of XML in the Creol language. For later work, we want to make XML fragments first class citizens in Creol and enhance the type system with types for XML schema to achieve a language capable of type-safe XML *programming*.

6.3 Part 3: Verification using a concrete, trace-based specification language

The third part of the work is presented in Papers #4 and #5. Here we introduce a new interface specification language, this time for Creol components. We obtain a specification-driven framework for testing Creol components by synchronizing the execution of a specification and an object. In Paper #4, we give the formal basis for the approach. In Paper #5, we show how to use Maude's state exploration to achieve *full* verification of a component with regard to its trace specification, and we present experimental results, which show the usefulness of our approach.

6.3.1 Paper #4

Title: Executable interface specifications for testing asynchronous Creol components
Authors: Immo Grabe, Martin Steffen, and Arild Braathen Torjusen
Publication: Research Report 375, Dept. of Informatics, Univ. of Oslo, July 2008 (revised May 2010) [GST08]. A shorter version with the same title but with the authors: Immo Grabe, Marcel Kyas, Martin Steffen, and Arild B. Torjusen was published in *Fundamentals of Software Engineering, Third IPM International Conference, FSEN 2009, Kish Island, Iran, April 15-17, 2009, Revised Selected Papers*, volume 5961 of *Lecture Notes in Computer Science*. Springer, 2010 [GKST10].

Summary
We propose and explore a formal approach for black-box testing of asynchronously communicating components in open environments. We use the Creol language for components and introduce an interface specification language for such components. To be able to formalize the observable behavior of Creol objects precisely, we introduce a calculus for Creol in the style of standard object calculi [AC96].

The behavior of an object in a particular execution is, at the interface, described by a sequence of communication labels (or communication events). The black-box behavior of an object can therefore be described by a set of *traces*, each consisting of a finite sequence of labels. The interface specification language is a concise trace language with prefix, choice and recursion, using communication labels as primitives.

In the specification language, a clean separation of concerns between interaction under the control of the component or coming from the environment is central. This leads to an assumption-commitment style specification of a component's behavior by defining the valid observable output behavior, assuming a certain scheduling of the input. Testing is done by synchronizing the execution of a specification and an object. Thus, input to the object is generated non-deterministically within the bounds of the specification, and at the same time, it is tested that the output behavior of the object conforms to the specification. Technically, the distinction between input and output interactions is made by formalizing well-formedness conditions on the specifications. Well-formedness enforces a syntactic distinction between input and output specifications and, in addition, assures that only "meaningful" traces, i.e., those corresponding to possible behavior, can be specified.

The specification language captures two crucial features of the interface behavior of Creol objects. First, the dynamic creation of objects and threads in Creol gives rise to dynamic scoping which

is reflected in the interface behavior by scope extrusion. The specification language allows expressing *freshness* of communicated object and thread references. Second, we take the asynchronous nature of communication in Creol into account by testing only up-to an appropriate notion of *observational equivalence*. The main contributions of the paper are:

Formalization We *formalize* the interface behavior of Creol plus a corresponding behavioral interface specification language. This gives the basis for testing active Creol objects, where a test environment can be simulated by execution of the specifications.

Implementation The existing Creol interpreter, implemented in rewriting logic and executable on the Maude platform, is extended with the implementation of the specification language. In a combined implementation, we synchronize communication between specification terms and Creol objects. This yields a specification-driven interpreter for testing asynchronous Creol components.

6.3.2 Paper #5

Title: Model testing asynchronously communicating objects using modulo AC rewriting
Authors: Olaf Owe, Martin Steffen, and Arild B. Torjusen
Publication: In *Proceedings of Model-Based Testing MBT'10 (ETAPS Satellite Workshop), March 2010.* To appear in *Electronic Notes in Theoretical Computer Science* [OST10].

Summary

In Paper #4, we introduced a formal approach for black-box specification-based testing of asynchronously communicating components in open environments together with an implementation of a testing framework. Here we show how to extend this approach to full verification of components. This is done by employing model checking via the *search* command of Maude.

Since our testing is based on behavior *observable* at the interface, the order of outgoing communication should not affect the test results. The operational semantics of the specification language considers this by treating certain reorderings of output events as observationally equivalent. This leads to a large increase in the reachable state space for the test cases. We investigate how our testing framework can handle this situation. Reordering of output events can be expressed by defining sequences of output events as *associative* and *commutative* (AC). We argue that our testing framework is especially well suited to implement this since, using the rewriting logic system Maude, associativity and commutativity can be declared using *equational attributes* [CDE+05] which allows efficient evaluation of such specifications. The main contributions of this paper are:

Verification We provide an implementation of the approach in the rewriter Maude and use its *search* functionality for state exploration (for rewriting modulo AC) for verification of components and investigate how the support for AC reasoning built into Maude contributes to state space reduction in verification of asynchronously communicating components.

Experimental results We present *experimental results* from using the Maude rewriting tool, which give empirical evidence of the benefits of our method. We compare, in two series of experiments, the influence on the state space of using Maude's AC support against explicit representation of all possible reorderings of output events (with the same semantics). Using AC

rewriting may considerably reduce the resource consumption when testing asynchronously communicating objects. AC rewriting significantly pays off in terms of time and the number of rewrites.

Chapter 7

Discussion

7.1 Contributions

The primary goal of the dissertation was stated in the introduction as

Overall goal. *Specification-based verification and testing of open distributed systems.*

To reach the goal, some more specific research goals related to Creol were identified. We will now evaluate the contribution of the dissertation towards these goals. We intended to investigate ways to specify Creol components, on the one hand in terms of behavior (Goal 1.1) and on the other in terms of data (Goal 1.2). The first, behavioral, approach turned out to be the most useful for developing methods for specification-based testing of Creol models and we have spent less time on the second approach, namely using XML for data level specification of interfaces.

7.1.1 Goal 1.1: Behavioral interface specifications

We have developed two different formalisms for interface level specification of behavior of object-oriented components communicating via asynchronous method calls. In Papers #1 and #2, we use the syntactic interface specifications of Creol as a starting point, and expand them with first order logic predicates over the history of the objects' observable communications. This allows for great flexibility in defining predicates since they may be given in a very abstract manner; it is not very programmer friendly, however, since it is less obvious from the abstract specification what behavior is actually specified. On the other hand, since the predicates are in first order logic, checking of them can be more directly implemented in rewriting logic.

Figure 7.1 gives an example of this first kind of specification. The client interface is an interface for nodes taking part in a distributed system for resource sharing, where nodes have an initial amount of local resources and may borrow from each other in order to perform required tasks. A node may only lend its own resources, borrowed resources may only be returned. The specification states that a given client may not lend out more resources than it got initially, and for every other client in the system it may not return more resources to a client than it has borrowed from that client (for further details see Paper #2).

In the Papers #4 and #5, we likewise use the syntactic interfaces of Creol but expand them with an *explicit* trace language. A specification in this language consists of *concrete* incoming or outgoing communication events, concrete in the sense that each event is specified, but not in the sense that

interface *Client*(init:*Nat*)
begin
with *Client*
  op borrow(a:*Nat* **out** b:*Nat*)
  op return(a:*Nat*)
  spec $0 \leq \mathit{lent}(\mathit{this}, h) \leq \mathit{init} \wedge \forall c : \mathit{Client} \cdot \mathit{lent}(c, h) \geq 0$
  where $\mathit{lent}(o, h) = \mathit{sum}((h/ \leftarrow o.\mathit{borrow}).b) - \mathit{sum}((h/ \rightarrow o.\mathit{return}).a)$
end

Figure 7.1: Abstract behavioral specification.

interface *Broker*
begin
with *Client*
  op getP(**in** item: *String* ; **out** price: *Int*)
with *Provider*
  op reg(**in** pr: *Provider*)
spec $=$ $n_{c1}\langle call\ b.getP(x)\rangle?\ .$
$(\ n_1\langle call\ p_1.getQ(x)\rangle!\ .\ n_2\langle call\ p_2.getQ(x)\rangle!\ )\ .$
$n_1\langle return(v_1)\rangle?\ .\ n_2\langle return(v_2)\rangle?\ .\ n_{c1}\langle return(v)\rangle!\ .\ \epsilon.$
end

Figure 7.2: Trace-based behavioral specification.

it contains concrete values for object references and method parameters. The specification language models the communication of the Creol components by using Creol-like communication labels as primitives.

An expression in the specification language has an operational semantics, and the language can be implemented. Indeed the main point of the method is that we implement the specification language and can execute a specification in parallel with the program to be tested. For an example of this second kind of specification, see Figure 7.2. The specified broker acts as an intermediary between a client and several providers of some service. A broker should, after being requested to do so by a client, query a fixed number of providers for a (price) quote and return an answer to the client with the best alternative found. Observe that this notation for specifications is closer to the programming language notation, using parametrized method calls and returns. This gives a more intuitive style for specification. The design goals for this specification language were that it should be executable, concise, and intuitive to use for someone knowing the Creol language.

7.1.2 Goal 1.2: Data-based interface specifications

We have extended Creol with types and operators for XML such that XML can be represented in Creol, and we have introduced functionality for validating XML against DTDs. This was done by extending the operational semantics of Creol through extending the functional sublanguage in Maude with sorts and constructors for XML data, and for XML schemas. We have implemented the DTD schema language in Maude by introducing regular expressions to describe document structures. With this as a basis, we have implemented an algorithm for validating XML documents against XML schemas. Thus, we get a data type for XML documents, which may be validated

```
interface LibraryServ
begin
with LibraryCl
    op getEntries(in query:QueryElemNd ; out result:ResultElemNd)
    where QueryElemNd =
      xmlSchema("query",
      elemDecl("query" , elemCt("title" | ("author" | ("editor" | "publisher"))))
      elemDecl("title", elemCt(PCDATA))
      elemDecl("author", elemCt(PCDATA))
      elemDecl("editor", elemCt(PCDATA))
      elemDecl("publisher", elemCt(PCDATA)), noAttDecl)
    where ResultElemNd =
      xmlSchema("result",
      elemDecl("result" , elemCt("book" | "err_result"))
      elemDecl("err_result", elemCt(PCDATA))
      elemDecl("book", elemCt("title" @ ("author"+ |"editor"+) @ "publisher" @ "price"))
      elemDecl("author", elemCt(PCDATA))
      elemDecl("editor", elemCt(PCDATA))
      elemDecl("title", elemCt(PCDATA))
      elemDecl("publisher", elemCt(PCDATA))
      elemDecl("price", elemCt(PCDATA))
end
```

Figure 7.3: Data-based interface specification using DTDs.

within Creol.

As a result, Creol applications may use XML documents for data storage and exchange. We have constructed a Creol example with a system consisting of a Library Server and a Client, where interface communication is verified by validating the exchanged XML data against DTDs (see Chapter 5 of Paper #2). This can be lifted to the level of interface specifications by adding DTDs to the interface declaration as exemplified in Figure 7.3. What we have done so far are only first steps towards integration of XML in Creol, as we have worked with simplified XML and DTD models.

7.1.3 Goal 1.3: Methods for verification of Creol models

The two behavioral interface specification languages contribute to two different approaches to attaining this Goal, and through this to the main Creol-specific research Goal 1. In both we use the specifications for simulation of environment behavior and as drivers in frameworks for testing components. We have described and implemented two different executable testing frameworks, thus achieving Goal 1.3.2. The implementations are described in detail in the research papers, and in Chapter 5. With regard to Goal 1.3.1, we have shown how to do model-based testing of Creol models by applying the ioco testing theory (see Subsection 4.1.5) to our setting of object-oriented asynchronous models. The underlying formal technicalities are presented in the papers; see also Chapter 6 for an overview and summary of the specific contributions of each.

The first testing method relies on metalevel strategies in rewriting logic to obtain an abstract environment for verification of Creol components. The technique used for monitoring and con-

trolling execution at the metalevel is quite general and though we here work with specifications that are method-based and object-oriented, the technique can be adapted to other kinds of executable rewriting logic specifications. An advantage of the metalevel technique is that the functionality needed for monitoring and for testing a component can be introduced transparently; there is no need to modify or annotate the test configuration.

In the second method, we develop a framework for testing Creol components by extending the existing Creol interpreter such that we may execute the synchronous parallel composition of a component with a specification acting as the environment, as described above in Section 5.2. With this approach, we likewise avoid the need for modification of the code being tested. The approach relies on a formalization of the interface behavior of Creol and a corresponding behavioral interface specification language.

Goal 1.3 mentions three properties we would like to have for the verification methods, namely that they should: (1) be automatic and supported by tools; (2) be compositional; and (3) tackle non-determinism. Both the methods that we have developed are tool-based and automatic: Given a specification and the component to test, verification is automatic by executing the frameworks. We have demonstrated this through the examples, which were designed for evaluating the implementations.

The methods we have used *support* compositionality by insisting on a black-box perspective and by relying on observable behavior only. We have only worked at the component level and have not addressed the issue of *compositionality* for the testing methods for Creol, i.e., how testing of components can be used as a basis for conclusions about these systems as a whole.

We have addressed the question of tackling non-determinism by investigating how the mechanisms for rewriting modulo associativity and commutativity built into the execution platform Maude can be used for more efficient verification by reducing the state space. When evaluating our approach by experimenting with the case studies we get evidence that using modulo AC rewriting enable us to cover more extensive test cases than we could do otherwise.

7.1.4 Verification and testing of open distributed systems

The primary goal of this dissertation is verification and testing of open distributed systems. We have explored the subject at the model level, through the Creol modeling language and have described methods for verification of Creol models. Two questions may be raised; first, how do the results generalize to verification and testing of open distributed systems in other modeling, and programming languages? Second, how can our methods for testing Creol *model* components be used for verification of program *code*?

To answer the first question; our methods for verification of Creol models are relevant for verification of open distributed systems in general. We have designed two different ways to formally specify object-oriented components in terms of interface behavior. We take a compositional view by restricting specifications to observable behavior only. By keeping a strict distinction between input and output, we get an assumption-commitment style description of behavior. We use the specification to simulate environment behavior by generating test input to components according to the assumptions and as an oracle for evaluation of the test output from the component. Thus, we show how simulation of environment behavior within the bounds of a specification can be combined with monitoring of the execution of the component to achieve testing for conformance

of component and specification.

The methods are implemented for Maude and Creol. In the first method for verification of components, we exploit the reflective character of rewriting logic. In the second, we rely on the fact that the Creol interpreter is implemented in Maude, which makes it amenable for modification to enable interaction with behavior specifications implemented in rewriting logic. We further use Maude's built-in support for rewriting modulo equations. Hence, the way we have concretely implemented the methods depends on the target modeling language, Creol, and the interpreter platform, Maude; and we show how to build test frameworks for Creol components on this basis. Nevertheless, the overall approach may be used in other settings, and applied to other languages to achieve assumption-commitment style descriptions of behavior that may be used for testing, adaptations would of course be necessary.

We have shown, in the setting of Creol, how to use interface specifications to simulate arbitrary environment behavior. In contrast to testing methods based on writing dummy code, or mock objects to simulate environment behavior, our methods avoid explicit programming of the behavior of objects to mimic the environment; depending on the type of specification language, we either deduce an alphabet of possible interactions from the abstract specification, or, with a concrete trace-based specification language, we generate communication events from the specification. These methods for using specifications could be applied to build drivers for testing components in other languages.

Another way to employ the results is, instead of applying the same methodology to build test drivers for other languages, to use the Maude dependent methods for verification of non-Maude components through facilities for external communication. Maude specifications may communicate with external objects through message passing over TCP sockets, or alternatively, through the InterOperability Platform (IOP) [MT05]. The IOP is an actor-based platform, which supports Maude modules through actors that function as wrappers for Maude specifications. Both methods allow Maude to communicate with external processes, and by using them, an environment simulated by Maude from specifications as described in this dissertation may communicate with actual components implemented in other languages on other platforms. We have not investigated these methods for using Maude models for concrete testing of actual components, this remains future work.

We have shown one way to achieve state space reduction using modulo AC rewriting in Maude. Since many forms of non-determinism inherent in distributed system can be formalized by means of associativity and commutativity, our approach to tackle non-determinism is relevant also in general for testing models or programs in an open distributed setting, regardless of whether the communication model is asynchronous, and for alternative definitions of observational equivalence.

As regards the second question, our methods for testing Creol models can be used in methodologies for testing implementations. In the context of the Credo project [Cre09], whose objective is modeling and analysis of evolutionary structures for distributed services, the article [G+09] describes how the Credo methodology can be used in software development. The Credo methodology combines the use of the executable data flow language Reo [Arb04] with Creol into a tool suite for modeling and analyzing highly reconfigurable distributed systems. With a peer-to-peer file sharing system as an example, it is shown how the approach to testing that we have described in Paper #4 can be used in a larger context. In software development using top-down design, we would start from a specification of the system using *behavioral interfaces*, which abstract from the details of the

components of the system. When Creol is used to model the functional behavior of the components, our method for verification can be used to check conformance between the model and its behavioral interface specification. Other testing techniques can be employed to further check for conformance between the Creol model and an *implementation* in a language like C or Java; such techniques are described in the articles [GAJS09] and [AGSS08] as mentioned in Subsection 4.2.1. Combining these methods yields a method for conformance testing of implementations against a specification, as devised in the Credo methodology.

7.2 Limitations

Some elements in this dissertation could have been more developed, and the following open questions reflect some of these shortcomings.

Compositionality As we have laid the foundation for compositionality by adopting a black-box perspective, but not explicitly addressed the issue of compositionality , this is future work. There are some research results on this subject, which may be adapted to our setting. The article [vdBRT03] deals with compositional testing in the context of ioco testing. The main question is what can be concluded concerning the whole system from testing of the components. It is shown in the article that with certain restrictions, the ioco testing theory is suitable for compositional testing in the sense that the integration of fully conformant components is guaranteed to be correct. The results in this article give pointers for further work for compositional testing of Creol models.

Data-based interface specifications using XML As regards further work on XML integration for Creol, I see two options: The first is to extend Creol with XML processing capabilities by introducing regular expression types in Creol and extend the type system to make XML fragments first class in the language. Adapting the type checking algorithms of languages like XDuce [HP03] and CDuce [BCF03] to Creol would allow for type safe programming with XML in Creol. As several XML processing languages exist, however, one should consider the benefits of doing this also for Creol.

The second option is to extend what we have done so far and implement the parts that we left out from the XML and DTD models in the first round, to achieve a complete integration of XML in Creol with validity checking for the full XML specification. This would enable Creol applications to use XML documents as data storage and exchange format. Doing this requires some Maude implementation, but is less challenging theoretically than the first option. Efficient implementations of XML validity checking are available in other languages, however, so instead of reimplementing this in Maude one might define an API in Creol and use external libraries for the validation. The route to take depends on the further development of the Creol language. The current implementation of Creol is in Maude for which little XML support exists; with a runtime environment for Creol programs based on languages as Java, C, or ML, XML validation could more easily be integrated.

Applications and case studies A weakness of the work presented in this dissertation is that we have not yet considered large-scale applications for our methods. The examples we have used for demonstration and experiments do constitute a *proof of concept* for the proposed methods, but

are small. Creol has successfully been used to model complex and highly dynamic communication systems, e.g. wireless sensor networks in [LBSG10], where the Ad hoc On-Demand Distance Vector (AODV) routing algorithm is used as a case study. Another example is ASK [Ask], an industrial size multi-threaded, asynchronous application for connecting people. A substantial part of ASK has been modeled in Creol [AGSS08]. Both models are large and quite complex. It would be interesting to apply our verification methods to one of these larger models. We have started work on applying our method for model-based testing of Creol models to the AODV model. We expect that the work with larger models will reveal a requirement for improved tool support, hence further development of tool support is a natural part of this work.

7.3 Future work

There are interesting challenges ahead that go beyond what have been treated in this dissertation.

Further development of the specification language Our specification language in Papers #4 and #5 could be developed further and made more expressive. As it is now a specification does not only denote one possible trace, but allows to specify sets of traces by using communication labels with variables that are matched with concrete values during the execution of the combined specification and component. A further step to increase the expressiveness of the language could be to allow symbolic expressions in the specification. The execution of the combined specification and component would operate on these symbolic expressions instead of matching variables in the specification with concrete values from the component. As mentioned, the technique of dynamic symbolic execution is applied to Creol *models*, and implemented in Maude in [GAJS09], and the results from this work might give a lead on how to proceed with symbolic execution for model *specifications*.

Different conformance relations We have investigated how testing based on the ioco relation may be applied to testing object-oriented asynchronous models. Several variants of the ioco relation exist [Tre08] and it would be interesting to go more in depth into how different relations of the ioco family could be used for testing in our context.

Combination with other verification methods Modular reasoning and verification techniques for Creol are studied by Johan Dovland in [Dov09]. His approach is to use formal reasoning based on Hoare logic. In the papers [DJO05, DJO07], he develops a compositional proof system with proof rules derived from Hoare rules of a sequential language. It would be interesting to see how our testing methods could be combined with Hoare logic-based verification and theorem proving.

In Paper #5, we use facilities for rewriting modulo AC built into the rewriting logic system Maude to achieve state space reduction in verification of Creol components. The paper [SMSdB04] treats verification of Rebeca models, and applies a range of abstraction techniques to alleviate the state-space explosion problem. Rebeca models have many similarities with Creol models (but also differences, see Subsection 2.1.2), and it would be interesting to investigate what further abstraction techniques could be applied to Creol models to achieve more efficient model checking.

New applications A new area for application of our testing framework is web services. In [CJO10] the authors show that a concurrent object language as Creol is well suited for high-level descriptions of service-oriented computing concepts. The article proposes to extend Creol with primitives for service-oriented computing, in particular for service publishing, discovery, and delegation. One direction for future work is to adapt the testing framework for Creol to enable testing of web service orchestration. To do this one would have to extend the testing framework with the new primitives as well. More developed XML support in Creol will also contribute to widening the application area for Creol as a modeling language for web services, since XML is central for some types of web services.

Further development of Creol—ABS In the recent EC project HATS [HAT], an abstract behavioral modeling language (ABS) [ABS10] is developed. The ABS language is an executable class-based object-oriented language. It is based on Creol, in particular on the concurrency model of Creol; ABS uses asynchronous method calls, and have underspecified local scheduling, which gives non-deterministic execution of models. ABS extends Creol in two interesting ways: First, by introducing user-defined abstract datatypes and a functional language over these types, which support pattern matching, and second, by introducing the concept of *concurrent object groups* (COG). COGs are based on the idea of CoBoxes[SPH08, SPH10] and is a generalization of the concurrency model of Creol. Instead of taking the single object as the unit of concurrency, acting as a monitor, one may take groups of objects, which share a computation resource. This means that there can be at most one activity running inside the *group*. In this dissertation, we have investigated different ways to introduce restrictions to execution of an underspecified model component through specifications. In particular, we have investigated how to use specifications for simulating environment behavior, and as drivers for testing. It is interesting to see how our work carries over to the generalized object group setting of the ABS language, and how our approach to verification of Creol models may be used for verification of models in the ABS language.

Bibliography

[Abr87] Samson Abramsky. Observation equivalence as a testing equivalence. *Theoretical Computer Science*, 53:225–241, 1987.

[ABS10] Report on the Core ABS Language and Methodology: Part A, April 2010. Deliverable D1.1A of project FP7-231620 (HATS), available at `http://www.cse.chalmers.se/research/hats/`.

[AC96] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Monographs in Computer Science. Springer-Verlag, 1996.

[Agh96] Gul A. Agha. Abstracting interaction patterns: A programming paradigm for open distributed systems. In Elie Najm and Jean-Bernard Stefani, editors, *Proceedings 1st IFIP International Conference on Formal Methods for Open Object-based Distributed Systems (FMOODS'96)*, pages 135–153, Paris, 1996. Chapman & Hall.

[AGSS08] Bernhard Aichernig, Andreas Griesmayer, Rudolf Schlatte, and Andries Stam. Modeling and testing multi-threaded asynchronous systems with Creol. In *Proceedings of the 2nd International Workshop on Harnessing Theories for Tool Support in Software (TTSS'08)*, Electronic Notes in Theoretical Computer Science. Elsevier, 2008.

[AL93] Martín Abadi and Leslie Lamport. Composing Specifications. *ACM Transactions on Programming Languages and Systems*, 15(1):73–132, January 1993.

[ÁMdBdRS02] Erika Ábrahám-Mumm, Frank S. de Boer, Willem-Paul de Roever, and Martin Steffen. Verification for Java's reentrant multithreading concept. In Mogens Nielsen and Uffe H. Engberg, editors, *Proceedings of the 5th International Conference on Foundations of Software Science and Computation Structures (FoSSaCS 2002)*, volume 2303 of *Lecture Notes in Computer Science*, pages 4–20. Springer-Verlag, April 2002.

[AMST97] Gul A. Agha, Ian A. Mason, Scott F. Smith, and Carolyn L. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7(1):1–72, January 1997.

[Arb04] Farhad Arbab. Reo: a channel-based coordination model for component composition. *Mathematical Structures in Computer Science*, 14(3):329–366, 2004.

[Ask] Ask community systems website. `http://www.ask-cs.com/`.

[Axe04] Eyvind W. Axelsen. A meta-level framework for recording and utilizing communication histories in Maude. Master's thesis, University of Oslo, August 2004.

[BB87] Tommaso Bolognesi and Ed Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14(1):25–59, January 1987.

[BBC+02] Jonathan P. Bowen, Kirill Bogdanov, John A. Clark, Mark Harman, Robert M. Hierons, and Paul Krause. FORTEST: formal methods and testing. In *Proceedings of the 26th Annual International Computer Software and Applications Conference (COMPSAC'02)*, pages 91–101, Los Alamitos, CA, USA, 2002. IEEE Computer Society.

[BCCZ99] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In Rance Cleaveland, editor, *Tools and Algorithms for Construction and Analysis of Systems, 5th International Conference, TACAS '99, Proceedings*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207. Springer-Verlag, 1999.

[BCF03] Veronique Benzaken, Guiseppe Castagna, and Alain Frisch. CDuce: An XML-centric general purpose language. In *Proceedings of the international conference on functional programming (ICFP'03)*, pages 51–63, 2003.

[BCM+92] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992.

[Bec99] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1999.

[Ber91] Gilles Bernot. Testing against formal specifications: A theoretical view. In Samson Abramsky and Tom S. E. Maibaum, editors, *Proceedings of the International Joint Conference on Theory and Practice of Software Development (TAPSOFT '91) Volume 2.*, volume 494 of *Lecture Notes in Computer Science*, pages 99–119. Springer-Verlag, April 1991.

[BFdV+99] Axel Belinfante, Jan Feenstra, Rene de Vries, Jan Tretmans, Nicolae Goga, Loe Feijs, Sjouke Mauw, and Lex Heerink. Formal test automation: A simple experiment. In Csopaki et al. [CDT99], pages 179–196.

[BGM91] Gilles Bernot, Marie-Claude Gaudel, and Bruno Marre. Software testing based on formal specifications: a theory and a tool. *IEEE Software Engineering Journal*, 6(6):387–405, November 1991.

[BH73] Per Brinch Hansen. *Operating system principles*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1973.

[BH77] Henry G. Baker and Carl Hewitt. The incremental garbage collection of processes. *ACM SIGPLAN Notices*, 12:55–59, 1977.

[BH99] Per Brinch Hansen. Java's insecure parallelism. *ACM SIGPLAN Notices*, 34:38–45, 1999.

[Bin00] Robert V. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison-Wesley, 2000.

[BJK+05] Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner, editors. *Model-Based Testing of Reactive Systems*, volume 3472 of *Lecture Notes in Computer Science*. Springer-Verlag, 2005.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. MIT Press, New York, 2008.

[Boo04] Grady Booch. *Object-Oriented Analysis and Design with Applications (3rd Edition)*. Addison-Wesley, 2004.

[BR70] John N. Buxton and Brian Randell, editors. *Software Engineering Techniques*. NATO Science Committee, 1970.

[Bru05] Stefan D. Bruda. Preorder relations. In Broy et al. [BJK+05], pages 117–149.

[BT01] Ed Brinksma and Jan Tretmans. Testing transition systems: An annotated bibliography. In *Modeling and Verification of Parallel Processes, 4th Summer School, MOVEP 2000*, volume 2067 of *Lecture Notes in Computer Science*, pages 187–195. Springer-Verlag, 2001.

[CDE+02] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and José F. Quesada. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 285:187–243, August 2002.

[CDE+05] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. *The Maude Manual (version 2.1.1)*. SRI International, Menlo Park, April 2005.

[CDT99] Gyula Csopaki, Sarolta Dibuz, and Katalin Tarnay, editors. *Proceedings of the IFIP TC6 12th International Workshop on Testing of Communicating Systems*. Kluwer Academic Publishers, 1999.

[CE82] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronisation skeletons using branching time temporal logic specifications. In Dexter Kozen, editor, *Proceedings of the Workshop on Logic of Programs 1981*, volume 131 of *Lecture Notes in Computer Science*. Springer-Verlag, 1982.

[CGP99] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, 1999.

[CJO10] Dave Clarke, Einar Broch Johnsen, and Olaf Owe. Concurrent objects à la carte. In Dennis Dams, Ulrich Hannemann, and Martin Steffen, editors, *Concurrency, Compositionality, and Correctness: Essays in Honor of Willem-Paul de Roever*, volume 5930 of *Lecture Notes in Computer Science*. Springer-Verlag, 2010.

[CKRW99] Pietro Cenciarelli, Alexander Knapp, Bernhard Reus, and Martin Wirsing. An event-based structural operational semantics of multi-threaded Java. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *Lecture Notes in Computer Science*, pages 157–200. Springer-Verlag, 1999.

[Cla00] Manuel Clavel. *Reflection in Rewriting Logic: Metalogical Foundations and Metaprogramming Applications*. CSLI Publications, Stanford, California, 2000.

[CM02] Manuel Clavel and José Meseguer. Reflection in conditional rewriting logic. *Theoretical Computer Science*, 285(2):245–288, 2002.

[Cre07] The Creol language. `http://heim.ifi.uio.no/creol`, 2007.

[Cre09] Credo (Modeling and analysis of evolutionary structured for distributed services). `http://www.cwi.nl/projects/credo`, 2009. European STREP Project in the Sixth Framework Programme, Priority 2, Information Society Technologies (2007–2009).

[Dij65] Edsger .W. Dijkstra. Programming considered as a human activity. In *Proceedings of the IFIP Congress 1965*, pages 213–217, 1965. EWD 117.

[Dij74] Edsger W. Dijkstra. On the role of scientific thought, August 1974. Published as EWD447: `http://www.cs.utexas.edu/users/EWD/ewd04xx/EWD447.PDF`.

[Dij75] Edsger W. Dijkstra. Guarded commands, nondeterminacy, and formal derivation. *Communications of the ACM*, 18(8):453–457, 1975.

[DJK+99] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz. Model-based testing in practice. In *Proceedings of the 1999 International Conference on Software Engineering, 1999*, pages 285–294, 1999.

[DJO05] Johan Dovland, Einar Broch Johnsen, and Olaf Owe. Verification of concurrent objects with asynchronous method calls. In *Proceedings of the IEEE International Conference on Software Science, Technology & Engineering(SwSTE'05)*, pages 141–150. IEEE Computer Society, February 2005.

[DJO07] Johan Dovland, Einar Broch Johnsen, and Olaf Owe. A compositional proof system for dynamic object systems. Res. Rep. 351, Dept. of Informatics, Univ. of Oslo, February 2007. revised March 2008.

[DMN68] Ole-Johan Dahl, Bjørn Myhrhaug, and Kristen Nygaard. (Simula 67) Common base language. Technical Report S-2, Norsk Regnesentral (Norwegian Computing Center), Oslo, Norway, May 1968.

[DMT00] Grit Denker, José Meseguer, and Carolyn Talcott. Formal specification and analysis of active networks and communication protocols: the Maude experience. 2000. Proceedings of DICEX 2000.

[DNH84] Rocco De Nicola and Matthew Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34:83–133, 1984.

[Dov09] Johan Dovland. Incremental reasoning about distributed object-oriented systems. Ph.D. dissertation, University of Oslo, 2009.

[FHT05] John Fitzgerald, Ian J. Hayes, and Andrzej Tarlecki, editors. *Proceedings of FM 2005*, volume 3582 of *Lecture Notes in Computer Science*. Springer-Verlag, 2005.

[FJJV97] Jean-Claude Fernandez, Claude Jard, Thierry Jéron, and César Viho. An experiment in automatic generation of test suites for protocols with verification technology. *Science of Computer Programming*, 29(1–2):123–146, July 1997.

[FTW05] Lars Frantzen, Jan Tretmans, and Tim A. C. Willemse. Test generation based on symbolic specifications. In J. Grabowski and B. Nielsen, editors, *Third International Workshop on Formal Approaches to Testing of Software, FATES 2004*, pages 1–15, 2005.

[G+09] Immo Grabe et al. Credo methodology. Modeling and analyzing a peer-to-peer system in Credo. In *3rd International Workshop on Harnessing Theories for Tool Support in Software*, 2009.

[GAJS09] Andreas Griesmayer, Bernhard Aichernig, Einar Broch Johnsen, and Rudolf Schlatte. Dynamic symbolic execution for testing distributed objects. In Catherine Dubois, editor, *Proceedings 3rd International Conference on Tests and Proofs (TAP'09)*, volume 5668 of *Lecture Notes in Computer Science*, pages 105–120. Springer-Verlag, July 2009.

[Gau95] Marie-Claude Gaudel. Testing can be formal, too. In Peter D. Mosses, Mogens Nielsen, and Michael I. Schwarzbach, editors, *Proceedings of TAPSOFT '95: Theory and Practice of Software Development, 6th International Joint Conference CAAP/FASE*, volume 915 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995.

[Gau05] Marie-Claude Gaudel. Formal methods and testing: Hypotheses, and correctness approximations. In Fitzgerald et al. [FHT05], pages 2–8.

[GJ98] Marie-Claude Gaudel and P. R. James. Testing algebraic data types and processes: a unifying theory. *Formal Aspects of Computing*, 10(5-6):436–451, 1998.

[GJSB00] James Gosling, Bill Joy, Guy L. Steele, and Gilad Bracha. *The Java Language Specification*. Addison-Wesley, Second edition, 2000.

[GKST10] Immo Grabe, Marcel Kyas, Martin Steffen, and Arild B. Torjusen. Executable interface specifications for testing asynchronous Creol components. In Farhad Arbab and Marjan Sirjani, editors, *Fundamentals of Software Engineering, Third IPM International Conference, FSEN 2009, Revised Selected Papers*, volume 5961 of *Lecture Notes in Computer Science*, pages 324–339. Springer-Verlag, 2010.

[Gla01] Rob van Glabbeek. The linear time–branching time spectrum I. In Jan A. Bergstra, Alban Ponse, and Scott A. Smolka, editors, *Handbook of Process Algebra*, pages 3–99. North-Holland, 2001.

[GM87] Joseph A. Goguen and José Meseguer. Order-sorted algebra solves the constructor-selector, multiple representation and coercion problems. *Information and Computation*, 103(1), June 1987.

[God91] Patrice Godefroid. Using partial orders to improve automatic verification methods. In Edmund M. Clarke and Robert P. Kurshan, editors, *Computer Aided Verification, 2nd International Workshop, CAV '90*, volume 531 of *Lecture Notes in Computer Science*, pages 176–449. Springer-Verlag, 1991.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implemementation*. Addison-Wesley, Reading, MA, 1983.

[Grü10] Andreas Grüner. Testing concurrent objects. Ph.D. dissertation (under preparation), University Leiden, 2010.

[GST08] Immo Grabe, Martin Steffen, and Arild Braathen Torjusen. Executable interface specifications for testing asynchronous Creol components. Technical Report 375, University of Oslo, Dept. of Computer Science, July 2008. revised May 2010.

[GWM+00] Joseph A. Goguen, Timothy Winkler, José Meseguer, Kokichi Futatsugi, and Jean-Pierre Jouannaud. Introducing OBJ. In *Software Engineering with OBJ: Algebraic Specification in Action*. Kluwer Academic Publishers, 2000.

[H+09] Robert M. Hierons et al. Using formal specifications to support testing. *ACM Computing Surveys*, 41(2):1–76, 2009.

[HAT] HATS (Highly Adaptable and Trustworthy Software using Formal Models). `http://www.hats-project.eu`. European IP Project in the Seventh Framework Programme (2009–2012).

[HBH08] Robert M. Hierons, Jonathan P. Bowen, and Mark Harman, editors. *Formal Methods and Testing*, volume 4949 of *Lecture Notes in Computer Science*. Springer-Verlag, 2008.

[HBS73] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *Third International Joint Conference On Artificial Intelligence (IJCAI 1973)*, August 1973.

[Hen88] Matthew Hennessy. *Algebraic Theory of Processes*. MIT Press, 1988.

[HM80] Matthew Hennessy and Robin Milner. On observing nondeterminism and concurrency. In Jaco W. de Bakker and Jan van Leeuwen, editors, *Proceedings of 7th Colloquium on Automata, Languages and Programming*, volume 85 of *Lecture Notes in Computer Science*, pages 299–309. Springer-Verlag, 1980.

[HM85] Matthew Hennessy and Robin Milner. Algebraic laws for nondeterminism and concurrency. *Journal of the ACM*, 32(1):137–161, 1985.

[Hoa69] Charles A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[Hoa74] Charles A. R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–557, 1974.

[Hoa85] Charles A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[Hoa96] Charles A. R. Hoare. How did software get so reliable without proof? In Marie-Claude Gaudel and Jim Woodcock, editors, *Industrial Benefit and Advances in Formal Methods (FME'96)*, volume 1051 of *Lecture Notes in Computer Science*, pages 1–17. Springer-Verlag, 1996.

[HP03] Haruo Hosoya and Benjamin C. Pierce. XDuce: A typed XML-processing language. *ACM Transactions on Internet Technology*, 3(2):117–148, 2003.

[HT99] J. He and K. Turner. Protocol-inspired hardware testing. In Csopaki et al. [CDT99], pages 131–147.

[HVP05] Haruo Hosoya, Jérôme Vouillon, and Benjamin C. Pierce. Regular expression types for XML. *ACM Transactions on Programming Languages and Systems*, 27(1), 2005.

[Int95] International Telecommunication Union. Open Distributed Processing - Reference Model parts 1–4. Technical report, ISO/IEC, Geneva, July 1995.

[JO05] Einar Broch Johnsen and Olaf Owe. A dynamic binding strategy for multiple inheritance and asynchronously communicating objects. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *Proceedings 3rd International Symposium on Formal Methods for Components and Objects (FMCO 2004)*, volume 3657 of *Lecture Notes in Computer Science*, pages 274–295. Springer-Verlag, 2005.

[JO07] Einar Broch Johnsen and Olaf Owe. An asynchronous communication model for distributed concurrent objects. *Software and Systems Modeling*, 6(1):35–58, March 2007.

[JOT06] Einar Broch Johnsen, Olaf Owe, and Arild B. Torjusen. Validating behavioral component interfaces in rewriting logic. In Farhad Arbab and Marjan Sirjani, editors, *Proceedings IPM International Workshop on Foundations of Software Engineering (FSEN 2005)*, volume 159 of *Electronic Notes in Theoretical Computer Science*, pages 187–204. Elsevier, May 2006.

[JOT08] Einar Broch Johnsen, Olaf Owe, and Arild B. Torjusen. Validating behavioral component interfaces in rewriting logic. *Fundamenta Informaticae*, 82(4):341–359, 2008.

[JOY06] Einar Broch Johnsen, Olaf Owe, and Ingrid Chieh Yu. Creol: A type-safe object-oriented model for distributed concurrent systems. *Theoretical Computer Science*, 365(1–2):23–66, November 2006.

[Kel76] Robert M. Keller. Formal verification of parallel programs. *Communications of the ACM*, 19(7):371–384, 1976.

[LBSG10] Wolfgang Leister, Joakim Bjørk, Rudolf Schlatte, and Andreas Griesmayer. Validation of Creol models for routing algorithms in wireless sensor networks. Technical Report 1024, Norsk Regnesentral (Norwegian Computing Center), February 2010.

[Lea99] Doug Lea. *Concurrent Programming in Java: Design Principles and Patterns*. Addison-Wesley, 2d edition, 1999.

[LGA96] Pascale Le Gall and Agnes Arnould. Formal specifications and test: Correctness and oracle. In Magne Haveraaen, Olaf Owe, and Ole-Johan Dahl, editors, *Recent Trends in Data Type Specification, 11th Workshop on Specification of Abstract Data Types Joint with the 8th COMPASS Workshop Selected Papers*, volume 1130 of *Lecture Notes in Computer Science*, pages 342–358. Springer-Verlag, 1996.

[LT98] Nancy A. Lynch and Mark R. Tuttle. An introduction to input/output automata. Technical Report MIT/LCS/TM 373, MIT Press, November 1998.

[LY96] David Lee and Mihalis Yannakakis. Principles and methods of testing finite state machines—A survey. In *Proceedings of the IEEE*, volume 84, pages 1090–1126, 1996.

[MC81] Jayadev Misra and K. Mani Chandy. Proofs of networks of processes. *IEEE Transactions on Software Engineering*, 7:417–426, 1981.

[Mea55] George H. Mealy. A method for synthesizing sequential circuits. *Bell System Technical Journal*, 34(5):1045–1079, 1955.

[Mes92] José Meseguer. Conditional rewriting as a unified model of concurrency. *Theoretical Computer Science*, pages 73–155, 1992.

[Mes07] Gerard Meszaros. *XUnit Test Patterns: Refactoring Test Code*. Addison-Wesley, 2007.

[MFC01] Tim Mackinnon, Steve Freeman, and Philip Craig. Endo-testing: Unit testing with mock objects. In Giancarlo Succi and Michele Marchesi, editors, *Extreme Programming Examined*, The XP Series, pages 287–301. Addison-Wesley, 2001.

[Mil80] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.

[Moo56] Edward F. Moore. Gedanken experiments on sequential machines. *Automata Studies*, pages 129–153, 1956.

[MT05] Ian A. Mason and Carolyn L. Talcott. IOP: The interoperability platform & IMaude: An interactive extension of Maude. *Electronic Notes in Theoretical Computer Science*, 117:315–333, 2005.

[Nel91] Greg Nelson, editor. *Systems Programming with Modula-3*. Prentice Hall, 1991.

[Nie87] Oscar Nierstrasz. Active objects in Hybrid. In *Object Oriented Programming: Systems, Languages, and Applications (OOPSLA) '87*, volume 22 of *ACM SIGPLAN Notices*, pages 243–253, 1987.

[Nie89] Oscar Nierstrasz. A survey of object-oriented concepts. In *Object-Oriented Concepts, Databases and Applications*, pages 3–21. ACM Press and Addison-Wesley, 1989.

[Nie92] Oscar Nierstrasz. A tour of Hybrid: A language for programming with active objects. In D. Mandrioli and B. Meyer, editors, *Advances in Object-Oriented Software Engineering*, pages 167–182. Prentice-Hall, 1992.

[NNH99] Flemming Nielson, Hanne-Riis Nielson, and Chris L. Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.

[OST10] Olaf Owe, Martin Steffen, and Arild Torjusen. Model testing asynchronously communicating objects using modulo AC rewriting. In *Proceedings of Model-Based Testing (MBT'10) (ETAPS Satellite Workshop)*, March 2010. Electronic Notes in Theoretical Computer Science.

[Pel93] D. Peled. All from one, one for all. In Costas Courcoubetis, editor, *Computer Aided Verification, 5th International Conference, CAV '93*, volume 697 of *Lecture Notes in Computer Science*, pages 409–423. Springer-Verlag, 1993.

[PL05] Alexander Pretschner and Martin Leucker. Model-based testing—a glossary. In Broy et al. [BJK+05], pages 607–609.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *SFCS '77: Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, pages 46–57. IEEE Computer Society, 1977.

[PP05a] Wolfgang Prenninger and Alexander Pretschner. Abstractions for model-based testing. In Mauro Pezzé, editor, *Proceedings of the International Workshop on Test and Analysis of Component Based Systems (TACoS 2004)*, volume 116, pages 59–71, 2005.

[PP05b] Alexander Pretschner and Jan Philipps. Methodological issues in model-based testing. In Broy et al. [BJK+05], pages 281–291.

[Pre05] Alexander Pretschner. Model-based testing in practice. In Fitzgerald et al. [FHT05], pages 537–541.

[PY02] Alexandre Petrenko and Nina Yevtushenko. Queued testing of transition systems with inputs and outputs. In Robert Hierons and Thierry Jéron, editors, *Formal Approaches to Testing of Software, FATES'02 workshop of CONCUR'02*, pages 79–93, 2002.

[SAdB+08] Rudolf Schlatte, Bernhard Aichernig, Frank S. de Boer, Andreas Griesmayer, and Einar Broch Johnsen. Testing concurrent objects with application-specific schedulers. In John Fitzgerald and Anne Haxthausen, editors, *International Colloquium on Theoretical Aspects of Computing (ICTAC)*, volume 5160 of *Lecture Notes in Computer Science*. Springer-Verlag, 2008.

[Sch10] Rudolf Schlatte. Passive testing with parallel object-oriented software models. Ph.D. dissertation, University Graz, 2010.

[SdBMS05] Marjan Sirjani, Frank S. de Boer, Ali Movaghar, and Amin Shali. Extended rebeca: a component-based actor language with synchronous message passing. In *Application of Concurrency to System Design, 2005. ACSD 2005. Fifth International Conference on*, pages 212–221, 7–9 2005.

[Sir06] Marjan Sirjani. Rebeca: Theory, applications, and tools. In *Proceedings of FMCO 2006*, Lecture Notes in Computer Science, pages 102–126. Springer-Verlag, 2006.

[SMSdB04] Marjan Sirjani, Ali Movaghar, Amin Shali, and Frank S. de Boer. Modeling and verification of reactive systems using Rebeca. *Fundamenta Informatica*, 63(4):385–410, Dec. 2004.

[SOA] SOA principles. `http://www.soaprinciples.com/`.

[SPH08] Jan Schäfer and Arnd Poetzsch-Heffter. CoBoxes: Unifying Active Objects and Structured Heaps. In *Formal Methods for Open Object-Based Distributed Systems (FMOODS 2008)*, pages 201–219, 2008.

[SPH10] Jan Schäfer and Arnd Poetzsch-Heffter. JCoBox: Generalizing active objects to concurrent components. In *24th European Conference on Object-Oriented Programming (ECOOP 2010)*, volume 6183 of *Lecture Notes in Computer Science*, pages 275–299. Springer-Verlag, June 2010.

[Std90] IEEE Std.610.12-1990. *IEEE Standard Glossary of Software Engineering Terminology*. The Institute of Electrical and Electronics Engineers: New York, 1990.

[Str86] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1986.

[TB03] Jan Tretmans and Ed Brinksma. TorX: Automated model based testing. In A. Hartman and K. Dussa-Ziegler, editors, *Proceedings of the 1st European Conference on Model-Driven Software Engineering*, pages 31–43, December 2003.

[TOS07a] Arild Torjusen, Olaf Owe, and Gerardo Schneider. Towards integration of XML in the Creol object-oriented language. Res. Rep. 365, Dept. of Informatics, Univ. of Oslo, October 2007. Revised February 2008.

[TOS07b] Arild Torjusen, Olaf Owe, and Gerardo Schneider. Towards integration of XML in the Creol object-oriented language. In F. E. Sanders, editor, *Proceedings of the Norsk Informatikkonferanse 2007 (NIK 2007)*, pages 107–111. Tapir Akademisk Forlag, 2007.

[TPvB97] Q.M. Tan, A. Petrenko, and Gregor v. Bochmann. Checking experiments with labeled transition systems for trace equivalence. In *Proceedings of the IFIP 10th International Workshop on Testing Communicating Systems (IWTCS'97)*, 1997.

[Tre96a] Jan Tretmans. Conformance testing with labelled transition systems: Implementation relations and test generation. *Computer Networks and ISDN Systems*, 29(1):49–79, 1996.

[Tre96b] Jan Tretmans. Test generation with inputs, outputs, and repetetive quiescence. *Software—Concepts and Tools*, 17(3):103–120, 1996.

[Tre99] Jan Tretmans. Testing concurrent systems: A formal approach. In Jos C.M. Baeten and Sjouke Mauw, editors, *CONCUR '99: Concurrency Theory 10th International Conference, Eindhoven, The Netherlands*, volume 1664 of *Lecture Notes in Computer Science*, pages 46–65. Springer-Verlag, August 1999.

[Tre08] Jan Tretmans. *Model Based Testing with Labelled Transition Systems*, pages 1–38. Volume 4949 of Hierons et al. [HBH08], 2008.

[Val92] Antti Valmari. A stubborn attack on state explosion. *Formal Methods in System Design*, pages 297–322, 1992.

[VCG+08] Margus Veanes, Colin Campbell, Wolfgang Grieskamp, Wolfram Schulte, Nikolai Tillmann, and Lev Nachmanson. Model-based testing of object-oriented reactive systems with spec explorer. In Hierons et al. [HBH08], pages 39–76.

[vdBRT03] Machiel van der Bijl, Arend Rensink, and Jan Tretmans. Compositional testing with ioco. In *Third International Workshop on Formal Approaches to Testing of Software, FATES 2003*, volume 2931 of *Lecture Notes in Computer Science*, pages 86–100. Springer-Verlag, 2003.

[W3C99] W3C (World Wide Web Consortium). *XML Path Language (XPath) Version 1.0*, 1999. W3C Recommendation 16 November 1999. `http://www.w3.org/TR/1999/REC-xpath-19991116`.

[W3C00] W3C. *Extensible Markup Language (XML) 1.0*. W3C (World Wide Web Consortium), 2 edition, 2000. W3C Recommendation 6 October 2000. `http://www.w3.org/TR/2000/REC-xml-20001006`.

[Yon90] Akinori Yonezawa. *ABCL: An Object-Oriented Concurrent System*. MIT Press, 1990.

[Zwi89] Job Zwiers. *Compositionality, Concurrency and Partial Correctness - Proof Theories for Networks of Processes, and Their Relationship*, volume 321 of *Lecture Notes in Computer Science*. Springer-Verlag, 1989.

Part II

Research papers

Chapter 8

Paper #1: Validating behavioral component interfaces in rewriting logic

Einar Broch Johnsen, Olaf Owe, and Arild B. Torjusen

Published in *Proceedings IPM International Workshop on Foundations of Software Engineering (FSEN 2005)*, volume 159 of *Electronic Notes in Theoretical Computer Science*, Elsevier, May 2006.[JOT06].

Validating Behavioral Component Interfaces in Rewriting Logic

Einar Broch Johnsen, Olaf Owe, and Arild B. Torjusen [1]

Department of Informatics, University of Oslo, Norway

Abstract

Many distributed applications can be understood in terms of components interacting in an open environment such as the Internet. Open environments are subject to change in unpredictable ways, as other applications may arrive, evolve, or disappear. In order to validate components in such environments, it can be useful to build a simulation environment which reflects this highly unpredictable behavior. In this paper, the validation of components with respect to behavioral interfaces is considered. Behavioral interfaces specify semantic requirements on the observable behavior of components, expressed in an assume-guarantee style. In our approach, a rewriting logic model is transparently extended with the history of all observable communication, and metalevel strategies are used to guide the simulation of environment behavior. Over-specification of the environment is avoided by allowing arbitrary environment behavior within the bounds of the assumption on observable behavior, while the component is validated with respect to the guarantee of the behavioral interface.

Key words: Validation, components, behavioral interfaces, simulation strategies, rewriting logic, meta-programming

1 Introduction

This paper suggests an application of rewriting logic [17] to test the behavior of software units in *open distributed environments* such as the Internet. An open environment is an environment in which various other software units exist, and little or no information about these units is available. A distributed environment is an environment in which communication is asynchronous. Reasoning in this setting is intrinsically difficult, partly due to the non-determinism caused by distribution, but more characteristically due to the unknown and evolving open environment.

It is a major challenge to predict the behavior of components evolving in open distributed environments, in order to ensure and maintain behavioral properties concerning safety, availability, quality of service, robustness, and fault tolerance. Formal approaches to system verification, such as Hoare logic, type checking,

[1] Email: einarj@ifi.uio.no, olaf@ifi.uio.no, aribraat@ifi.uio.no

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

and model checking, depend on knowing the implementation details of the system components, including those in the open environment. Approaches based on testing simulate an environment in which the system can be subjected to test runs. In contrast to verification methods, testing cannot generally ensure that components are always well-behaved, but testing may still give revealing insights into a component's behavior. However, the problem of conformance testing for software units in open distributed environments is not resolved [25]. This paper shows how open environments can be mimicked by underspecified formal descriptions based on *observable behavior* in order to validate the behavior of software units in open distributed environments at the modeling level. Model-based testing in the early development stages makes the testing process more effective [19].

Object orientation is the leading framework for concurrent and distributed systems, recommended by the RM-ODP [12] and used in, e.g., .Net and Corba. In this paper, we model distributed components by objects which asynchronously exchange messages. The models are executable in the rewriting logic system Maude [4], which has facilities for simulation, model checking, and verification. To allow black-box validation, we use requirement specifications in terms of observable behavior. Observable behavior is specified using *behavioral interfaces* [13,14] which describe component services available to the environment.

This paper defines an executable framework for validating the observable behavior of models in the open distributed setting. For this purpose, behavioral interfaces are captured in rewriting logic and combined with a standard rewriting logic model of asynchronously communicating objects. Furthermore, the executable platform in Maude is extended with validation facilities in a transparent way. Rewriting logic is *reflective* [3,5] in a mathematically precise manner: it is possible to reason formally about reflective rewriting inside rewriting logic itself, and to execute reflective specifications at the Maude *metalevel*. The use of reflection is essential to our approach, allowing for guided search and system monitoring in a modular, composable, and hierarchical way. Reflection may be used to define execution strategies for an executable object model, for example a *nondeterministic* execution strategy is proposed in [15]. Reflective specifications support a layered architecture where several specifications may be given at each level. Reflection can be used to extend a system model with, e.g., logging facilities [24]. In this paper, we transparently extend an executable specification with its history of observable communications at the metalevel, and define execution strategies at the metalevel which are guided by requirements on the communication history. One strategy is used to mimic open environments and another to test the executable model. The two strategies are combined in order to enable an assume-guarantee style model-based testing of components with respect to their behavioral interfaces.

Paper overview: Sect. 2 presents a formalism for behavioral interfaces. Sect. 3 presents rewriting logic and the Maude tool. Sect. 4 develops metalevel strategies for monitoring and testing executable Maude models. A strategy for simulation of open environments is presented in Sect. 5 and it is shown how this can be utilized in a test scenario. Sect. 6 discusses related and future work.

2 Behavioral Interfaces

An open distributed system (ODS) can be represented by components or objects that run in parallel and communicate asynchronously by means of remote method calls. The implementation details of the components may be unknown, in which case reasoning must rely on abstract specifications of the system's components. We assume that components come equipped with *behavioral interfaces* that instruct us on how to use them. As a component may be used for multiple purposes, it can come equipped with *multiple* interfaces. This section presents a formalism for viewpoints based on a notion of generic interface with behavioral requirements, restricted to safety aspects. For further details about this work, see [13,14].

Black-box specifications of concurrent components may be expressed in terms of *observable behavior*, i.e., the time sequence of input and output to the components. This fits well with the notion of encapsulation; only visible operations are considered at the specification level. An execution can be represented by a sequence of communication events, which is infinite in the case of non-terminating executions. However, infinite sequences are not easy to reason about. To avoid infinite sequences, specifications may be expressed in terms of the finite initial segments of the executions, capturing the abstract states of components during execution. These sequences are commonly referred to as histories [6] or traces [11]. Prefix-closed sets of executions express safety properties in the sense of Alpern and Schneider [1].

Finite sequences. We consider an abstract data type $\mathsf{Seq}[T]$ of finite sequences parameterized by a type T. Functions over sequences will be defined by means of convergent sets of equations, using the empty sequence, ε, and right append, $\_;\_ : \mathsf{Seq}[T] \times T \to \mathsf{Seq}[T]$, as sequence constructors. We let "_" denote argument positions of functions with mix-fix notation.

We define projection, $\_/\_ : \mathsf{Seq}[T] \times \mathsf{Set}[T] \to \mathsf{Seq}[T]$, and an "ends with" relation, $\_\mathbf{ew}\_ : \mathsf{Seq}[T] \times \mathsf{Set}[T] \to \mathsf{Bool}$, using one equation for each constructor case:

$$\varepsilon/S = \varepsilon \qquad \varepsilon \ \mathbf{ew}\ S = \mathit{false}$$
$$(t;x)/S = \mathbf{if}\ x \in S\ \mathbf{then}\ (t/S);x\ \mathbf{else}\ t/S \qquad (t;x)\ \mathbf{ew}\ S = x \in S$$

The notation $\#t$ denotes the length of a sequence t and is defined in a similar way.

2.1 Semantics

Let Ob be an unbounded set of object identifiers. Let Data be a set of data values, including Ob. In this paper, we conventionally let $o_1, o_2 \in \mathsf{Ob}$. A *communication event* has the form

$$\mathit{msg}\ \mathbf{from}\ o_1\ \mathbf{to}\ o_2$$

where *msg* consists of Data. This term is considered an *output event* of o_1 and an *input event* of o_2. For observable events, o_1 and o_2 are distinct. The sets of observable input and output events of an object o are denoted IN_o and OUT_o, respectively, and are by definition disjoint. Their union is denoted $INOUT_o$.

An *alphabet* for an object o is a subset of $INOUT_o$. An alphabet of o may cover certain aspects of the communication of o. In the next section we introduce syntax for statically defined alphabets. A *trace set* $\mathcal{T}_\alpha \subseteq \mathsf{Seq}[\alpha]$ is a prefix-closed set of well-formed sequences.

Definition 2.1 A *specification* Γ is a triple $\langle o, \alpha, \mathcal{T} \rangle$ where (1) $o \in \mathsf{Ob}$ is an object identifier, (2) α is a possibly infinite alphabet for o, and (3) $\mathcal{T}$ is a trace set over α.

For any specification Γ, we can derive a *communication environment* $\mathcal{E}(\Gamma)$ of objects communicating with the object of Γ. In an ODS setting, we generally think of the communication environment as unbounded. Since the specification Γ does not need to cover all aspects of the behavior of o, we say that Γ is an *interface specification* (of o).

In the following we consider object-oriented distributed systems where communication is achieved through remote methods calls. In order to achieve asynchronous communication, we model a method call through two events: the event representing the initiation of a call, and the event representing its completion. Let Mtd be an unbounded set of method names, and let $m \in \mathsf{Mtd}$. For a call by o_1 to method m of o_2, the initiation event is generated by the caller o_1 and is represented by $invoc(m)$ **from** o_1 **to** o_2, and the completion event is generated by the callee o_2 and represented by $comp(m)$ **from** o_2 **to** o_1. To simplify the exposition, we abstract from parameter values in this paper. In order to increase readability, we represent these events by $o_1 \rightarrow o_2.m$ and $o_1 \leftarrow o_2.m$, respectively.

As we consider asynchronously communicating objects, a caller may communicate while (passively) waiting for a completion and a callee may communicate while performing a method. Consequently, other events can be observed in between the initiation and completion of any given call. When we consider the history of observable behavior, every completion event must be preceded by a corresponding invocation, which gives rise to the following notion of well-formedness for communication histories:

$$\begin{aligned} wf(\varepsilon) &= true \\ wf(t;(o \rightarrow o'.m)) &= wf(t) \\ wf(t;(o \leftarrow o'.m)) &= wf(t) \wedge \#(t/o \rightarrow o'.m) \geq \#(t/o \leftarrow o'.m) \end{aligned}$$

where $\#(t/o \rightarrow o'.m)$ is the length of the trace t restricted to invocation events of the method m from o to o', and similarly for completion events.

Definition 2.2 A specification $\langle o, \alpha, \mathcal{T} \rangle$ of o *refines* another specification $\langle o, \alpha', \mathcal{T}' \rangle$ of o if $\alpha' \subseteq \alpha$ and $\forall t \in \mathcal{T} \,.\, t/\alpha' \in \mathcal{T}'$.

Thus, refinement corresponds to the subset relation on projected trace sets in the sense that $\{t/\alpha' \mid t \in \mathcal{T}\} \subseteq \mathcal{T}'$. Note that a specification may refine several specifications with (partially) disjoint alphabets. The composition of specifications may be introduced to define partial components or system aspects in the sense of distributed services [13,14].

2.2 Syntax

Interface specifications may be given in a generic manner. Generic specifications are referred to as *behavioral interfaces*. An object may support a number of interfaces. As Maude does not provide a syntax for specification of observable behavior, statically defined alphabets, nor methods (not even with Full Maude), we introduce a syntax for observable behavior by means of object-oriented interfaces:

interface F (⟨context parameters⟩)
  inherits $F_1, F_2, \ldots, F_m$
begin
with *cointerface*
    op $m_1(\ldots)$
      ...
    op $m_n(\ldots)$
  spec <formula on local trace>
  where <auxiliary function definitions>
end

Interfaces can have context parameters, which typically describe the minimal environment, representing static links needed by objects that support the interface. An initiation and a completion event is associated with each method declaration (ranging over method parameters, which are ignored in this paper). In the specification formula, the keyword "*this*" denotes the object supporting the interface.

Mutual dependency. Let objects be typed by interfaces. By identifying a type for the caller, the *cointerface*, we restrict the objects that may call the methods of this interface, while allowing *this* object to call cointerface methods. This opens up for interaction with a caller during execution of a method. In an implementation language, access to the *caller* may be provided by an explicit parameter as in Maude, or implicitly as in Creol [15]. Cointerfaces give strong typing in an asynchronous setting. Semantically a cointerface declaration augments the alphabet of the interface, as events related to cointerface methods are added.

Inheritance. Multiple inheritance is allowed for interfaces, but cyclic inheritance graphs are not allowed. In a subinterface, additional methods and behavioral constraints can be declared. A cointerface restriction applies to the locally declared methods. If an interface F is declared with an inheritance clause, the alphabets of the super-interfaces are included in the alphabet of F. Trace sets are inherited by intersection, when restricted to the relevant alphabets of the super-interfaces. Thus, an interface will always refine its super-interfaces.

Definition 2.3 The *interface alphabet* of an object o with respect to an interface F, denoted $\alpha_{o:F}$, is defined as the set of events of the form

(i) $invoc(m)$ **from** o' **to** o and $comp(m)$ **from** o **to** o' for m declared in F,

(ii) $invoc(m)$ **from** o **to** o' and $comp(m)$ **from** o' **to** o for m declared in (or inherited by) the cointerface, and

(iii) any event in $\alpha_{o:F'}$ where F' is a super-interface of F.

Definition 2.4 Let $F, F_1, \ldots, F_n$ be interfaces with corresponding specification predicates $P, P_1, \ldots, P_n$ and let h range over histories. If F inherits $F_1, \ldots, F_n$, the *interface specification* of F is the conjunction $P(h) \wedge P_1(h/\alpha_{this:F_1}) \wedge \ldots \wedge P_n(h/\alpha_{this:F_n})$.

Assume-guarantee predicates. In ODS, the environment in which an object exists is subject to change, and specifications are relative to an assumed behavior of the environment. We adapt the assume-guarantee specification style [16] to the setting of observable behavior. Assumptions should express restrictions on the inputs and guarantees on the outputs. However, it is often difficult to formulate assumptions and guarantees separately, since requirements to outputs may depend on earlier input, and requirements to inputs may depend on earlier output. Instead we use a single predicate P which relates input and output events, and extract an assumption part and a guarantee part from P:

Definition 2.5 Let *IN* and *OUT* denote the sets of input and output events for *this* interface. An *assume-guarantee* predicate is derived from the specification **spec** $P(h)$, where the assumption part A and the guarantee part G are defined by the equations

$$\begin{aligned} A(\varepsilon) &= \mathit{true} \\ A(h;x) &= A(h) \wedge (x \in \mathit{IN} \wedge P(h) \Rightarrow P(h;x)) \\ G(\varepsilon) &= \mathit{true} \\ G(h;x) &= G(h) \wedge (A(h;x) \Rightarrow P(h;x)) \end{aligned}$$

The trace set given by the specification **spec** $P(h)$ is $\{h \mid G(h)\}$.

Note that both sets $\{h \mid G(h)\}$ and $\{h \mid A(h)\}$ are prefix-closed, and that their intersection is the largest (prefix-closed) trace set contained in $\{h \mid P(h)\}$.

Assumptions are the responsibility of the objects in the environment. The assumption part ensures that each input is acceptable, assuming no earlier violation. Guarantees are the responsibility of the object supporting the interface; they are guaranteed when the assumption holds. The guarantee part ensures that each output is acceptable, assuming the assumption holds. Thus, an actual environment is required to refine the trace set given by A, and an implementation of the interface is required to refine the trace set given by G.

2.3 Example: A Minimal Interface

Behavioral interfaces are illustrated through the example of the dining philosophers. A table object informs a philosopher of the identity of the philosopher's left neighbor and provides units of food. A philosopher may borrow and return its neighbor's chopstick. Interaction between the philosophers and the table is restricted by interfaces. This results in a clear distinction between internal methods and methods externally available to other objects typed by the *cointerface*. Here, each philosopher owns one chopstick and must borrow another from a neighbor before eating. Hence, philosophers have both active and passive behavior. Strong typing and cointerfaces guarantee that only philosophers may call the methods *borrowStick* and *returnStick*.

```
interface Phil                  interface Table
begin                           begin
with Phil                       with Phil
  op borrowStick                  op seat(out neighbor:Phil)
  op returnStick                  op eat
  ⟨specification⟩               end
end
```

Denote by *caller* an arbitrary *Phil* object in the environment of *this Phil* object, as required by the cointerface. The alphabet of *Phil* is given by the events:

$$\begin{array}{ll} caller \rightarrow this.borrowStick & caller \leftarrow this.borrowStick \\ this \rightarrow caller.borrowStick & this \leftarrow caller.borrowStick \end{array}$$

and similar events for *returnStick*. We define the following specification in *Phil*:

$$\begin{array}{l} \textbf{spec} \quad 0 \leq lent(h) \leq 1 \wedge 0 \leq borrowed(h) + requested(h) \leq 1 \\ \textbf{where}\ lent(h) = \#(h/ \leftarrow this.borrowStick) - \#(h/ \rightarrow this.returnStick) \\ \quad borrowed(h) = \#(h/this \leftarrow borrowStick) - \#(h/this \rightarrow returnStick) \\ \quad requested(h) = \#(h/this \rightarrow borrowStick) - \#(h/this \leftarrow borrowStick) \end{array}$$

Here, *lent* captures the number of sticks lent to neighbors, *borrowed* the number of sticks the object has borrowed from its neighbors, and *requested* captures the number of unfulfilled borrow requests. The three functions are defined in terms of the history of observable behavior up to present time. The specification implies that a single boolean variable suffices to keep track of sticks given away. Thus, the assumption part of the specification reduces to

$$A_{Phil}(h;x) = A_{Phil}(h) \wedge (x \in \{\rightarrow this.returnStick\} \ \Rightarrow lent(h) > 0)$$

stating that the environment may not return more sticks than it has borrowed.

The two interfaces above are connected by introducing an interface *EatingPhil*, inheriting *Phil* and with a *Table* as a parameter, thereby providing initial environmental knowledge. The specification of *Phil* is strengthened by requiring that a philosopher must have two sticks to eat:

$$\begin{array}{l} \textbf{interface}\ EatingPhil(table : Table)\ \textbf{inherit}\ Phil \\ \textbf{begin} \\ \quad \textbf{spec} \quad eating(h) \Rightarrow lent(h) = 0 \wedge borrowed(h) = 1 \\ \quad \textbf{where}\ eating(h) = \#(h/this \leftarrow eat) > \#(h/this \rightarrow eat) \\ \textbf{end} \end{array}$$

Here, *eating* is true when *this* object is capable of eating. This interface does not strengthen the assumption inherited from *Phil*, i.e., $A_{EatingPhil}(h) = A_{Phil}(h) = \forall h' \leq h \cdot lent(h') \geq 0$.

3 Rewriting Logic and Maude

This section gives a brief introduction to rewriting logic [17] and Maude [4]. A rewrite theory is a 4-tuple $\mathcal{R} = (\Sigma, E, L, R)$, where the signature Σ defines the function symbols of the language, E defines equations between terms, L is a set of labels, and R is a set of labeled rewrite rules. From a computational viewpoint, a rewrite rule $t \longrightarrow t'$ may be interpreted as a *local transition rule* allowing an instance of the pattern t to evolve into the corresponding instance of the pattern t'. Rewrite rules apply to fragments of a state configuration. If rewrite rules may be applied to non-overlapping fragments of the configuration, the transitions may be performed in parallel. Consequently, rewriting logic (RL) is a logic which easily captures concurrent change. A number of concurrency models have been successfully represented in RL [4,17], including Petri nets, CCS, Actors, and Unity.

Informally, a state configuration in RL is a multiset of terms of given types, specified in (membership) equational logic (Σ, E), the functional sublanguage of RL which supports algebraic specification in the OBJ [10] style. Memberships express that a term belongs to a given sort. When modeling computational systems, configurations may include different system components modeled by terms of the different types defined in the equational logic. An RL object is a term $\langle O : C \mid a_1 : v_1, \ldots, a_n : v_n \rangle$, where O is the object's identifier, C is its class, the a_i's are the names of the object's attributes, and the v_i's are the corresponding values [4].

RL extends algebraic specification techniques with rewrite rules to capture the dynamic behavior of a system, supplementing the equations defining the term language. Assuming that all terms can be reduced to normal form, rewrite rules transform terms modulo the equations of E. Rewrite rules may have a condition (a conjunction of rewrites, equations, and memberships) which must hold for the main rule to apply. Each rule describes how a part of a configuration can evolve in one transition step:

$$\textbf{rl}\ [\text{label}] : \textit{subconfiguration} \longrightarrow \textit{subconfiguration}$$
$$\textbf{crl}\ [\text{label}] : \textit{subconfiguration} \longrightarrow \textit{subconfiguration}\ \textbf{if}\ \textit{condition}$$

An unconditional rule with an *if-then-else* expression as the right hand side may alternatively be given as two complementary conditional rules. Rules in RL may be formulated at a high level of abstraction, closely resembling a compositional operational semantics [18]. The Maude system supports analysis of RL specifications.

3.1 Reflection and The Maude Metalevel

Rewriting logic is reflective in the sense that there is a finitely presented rewrite theory $\mathcal{U}$ that is *universal*: any finitely presented rewrite theory $\mathcal{R}$ (including $\mathcal{U}$ itself) can be represented in $\mathcal{U}$. Let C and C' be configurations and $\mathcal{R}$ be a set of rewrite rules. We write $\mathcal{R} \vdash C \rightarrow C'$ to express that C may be rewritten to C' in the rewrite theory $\mathcal{R}$. Informally, a configuration C and the set $\mathcal{R}$ of rewrite rules of a specification in RL may be represented by terms $\overline{C}$ and $\overline{\mathcal{R}}$ at the metalevel. Using this notation, we have the equivalence [3]:

rl [req-stick] : $\langle X : Ob \,|\, hungry : true, myS : yes, nbrS : no, : nbr : Y\rangle \quad \longrightarrow$
$\langle X : Ob \,|\, hungry : true, myS : yes, nbrS : req, nbr : Y\rangle \; (invoc('borrowStick) \; from \; X \; to \; Y)$.

rl [borrow] : $\langle X : Ob \,|\, hungry : false, myS : yes, nbrS : s, nbr : Y\rangle$
$(invoc('borrowStick) \; from \; Z \; to \; X) \quad \longrightarrow$
$\langle X : Ob \,|\, hungry : false, myS : no, nbrS : s, nbr : Y\rangle \; (comp('borrowStick) \; from \; X \; to \; Z)$.

rl [rcv-stick] : $\langle X : Ob \,|\, hungry : true, myS : yes, nbrS : req, nbr : Y\rangle$
$(comp('borrowStick) \; from \; Y \; to \; X) \quad \longrightarrow$
$\langle X : Ob \,|\, hungry : true, myS : yes, nbrS : yes, nbr : Y\rangle$.

rl [eat-req] : $\langle X : Ob \,|\, hungry : true, myS : yes, nbrS : yes, nbr : Y\rangle \quad \longrightarrow$
$\langle X : Ob \,|\, hungry : true, myS : yes, nbrS : yes, nbr : Y\rangle \; (invoc('eat) \; from \; X \; to \; 'table)$.

rl [eat] : $\langle X : Ob \,|\, hungry : true, myS : yes, nbrS : yes, nbr : Y\rangle$
$(comp('eat) \; from \; 'table \; to \; X) \quad \longrightarrow$
$\langle X : Ob \,|\, hungry : false, myS : yes, nbrS : no, nbr : Y\rangle \; (invoc('returnStick) \; from \; X \; to \; Y)$.

Figure 1. Rewrite rules capturing philosopher behavior.

$$\mathcal{R} \vdash C \to C' \Leftrightarrow \mathcal{U} \vdash \langle \overline{\mathcal{R}}, \overline{C}\rangle \to \langle \overline{\mathcal{R}}, \overline{C'}\rangle,$$

which states that if a term C can be rewritten to a term C' in the rewrite theory $\mathcal{R}$, then the meta-representation of C in $\mathcal{R}$, $\langle \overline{\mathcal{R}}, \overline{C}\rangle$, can be rewritten to the meta-representation of C' in $\mathcal{R}$, $\langle \overline{\mathcal{R}}, \overline{C'}\rangle$, in the universal rewrite theory $\mathcal{U}$, and vice versa. Maude includes facilities to meta-represent a rewrite theory $\mathcal{R}$ and to apply rules from $\mathcal{R}$ to the meta-representation of a term C by so-called *descent functions*.

Metalevel rewrite rules may be used to select which rule from $\mathcal{R}$ to apply to which subterm of C. This is done by defining an interpreter function which takes as arguments a finitely presented rewrite theory $\mathcal{R}$, a term C, and a deterministic strategy S. Metalevel rewrite rules may further be used to modify a configuration or the rule set of a rewrite theory. Hence, metalevel rewriting can be used as a wrapper around a rewrite theory $\mathcal{R}$ in order to abstractly mimic a more elaborate rewrite theory $\mathcal{R}'$ extending $\mathcal{R}$. Further details on the theory and the use of reflection in RL and Maude may be found in [3,4,5].

3.2 Example: Implementation of the Philosophers

We introduce a Maude specification which implements the *EatingPhil* specification given in Sect. 2.3. Let O be a variable ranging over Ob, a philosopher object is defined as a RL object $\langle O : Ob \,|\, hungry : \_, myS : \_, nbrS : \_, nbr : \_\rangle$. The Boolean attribute *hungry* indicates whether the philosopher is hungry, the attributes *myS* and *nbrS* indicate the status of its chopsticks (*yes*,*no*,*req*), used to impose synchronization constraints on the specification, and *nbr* identifies the neighbor.

The philosopher interacts asynchronously with the environment by message passing. Internal actions are represented by a philosopher (asynchronously) passing messages to himself. A selection of rules from the specification is given in Fig. 1.

```
crl [exec-monitor] :
  ⟨M : MetaRep | curTerm : T, curModule : MOD, labels : L LS, failedRules : FR⟩
  ⟨History : H⟩                    ⟶
  if RES :: Result4Tuple then
    ⟨M : MetaRep | curTerm : getTerm(RES),curModule : MOD, labels : LS L,
                   failedRules : nil⟩
    ⟨History : H ; getNewMessages(T, getTerm(RES), MOD, H)⟩
  else
    ⟨M : MetaRep | curTerm : T, curModule : MOD, labels : LS L, failedRules : FR L⟩
    ⟨History : H⟩ fi
if  RES := metaXapply([MOD], T, L, none, 0, unbounded, 0) ∧ #FR ≤ #LS.
```

Figure 2. The metalevel rewrite strategy $\mathcal{S}_{monitor}$ records the communication history. The membership *RES :: Result4Tuple* expresses that the rewrite bound to *RES* succeeds, using a condition of the form *RES* := *term* to bind a term to *RES*.

4 Monitoring and Testing Executable Models

The observable behavior of an executable model can be monitored by recording the *communication history* from an execution of the model: This can be done *transparently* with the aid of the Maude metalevel without modifying the original specification. We can further test that the execution conforms to the behavioral specification of the model by defining metalevel predicates that operate on the recorded history and block execution if a violation occurs.

To execute a specification at the metalevel, we develop a custom *strategy*; i.e., rewrite rules which apply to the meta-representation of the model. Thus the current state may be inspected in-between rewrites. This enables us to record a communication history while executing a specification: We can check whether the application of a rewrite rule results in the emission of a new message by comparing the metalevel representations of the configuration before and after the rule application.

The object $\langle M : MetaRep \mid curTerm : \_, curModule : \_, labels : \_, failedRules : \_ \rangle$ is used to store the information needed to control consecutive metalevel rewrites. *curTerm* contains the meta-representation of the current configuration, *curModule* is the meta-representation of the name of the object-level module in which the rewrites will be performed, *labels* is a list of rule labels from this module, and *failedRules* contains a list of labels for rules that are not applicable to *curTerm*.

The object $\langle History : \_ \rangle$ has an attribute h which contains the actual communication history recorded at runtime as a message list. This object is distinct from the objects of the object-level model and is consequently not modified by nor needed for the application of any rewrite rule from the object-level specification.

The custom strategy $\mathcal{S}_{monitor}$ is implemented as a conditional rewrite rule $exec : MetaRep \times History \rightarrow MetaRep \times History$ (see Fig. 2). The actual rewriting is done by the built-in Maude function *metaXapply*, which returns a tuple from which the rewritten term is obtained using *getTerm*. Note that whitespace in Maude denotes

list concatenation: If L is a label and LS is a list of labels, then $L\ LS$ is a non-empty list of labels. The strategy applies rules from the *labels* list to the metalevel configuration in *curTerm* in a round-robin fashion. (A position-fair strategy for random rule selection based on a pseudo-random number generator is given in [15].) If no rule is applicable, the execution will terminate. The auxiliary function *getNewMessages* compares the term T to the new system configuration, i.e., the result of applying the rule labeled L to T. If there are new communication messages in the new system configuration, the attribute h of the history object is extended with the new messages. If there are several new messages, these are caused by concurrent actions and may therefore be added to the history in an arbitrary order.

The strategy $\mathcal{S}_{test}$ is defined by extending $\mathcal{S}_{monitor}$ with functionality to check whether a given rule application will lead to an illegal state, as specified by a predicate parameter. We consider predicates on communication histories as defined by behavioral interfaces. To obtain a compositional system, the predicate on the global history will be formulated as the conjunction of the requirement specifications of a number of behavioral interfaces, possibly associated with different objects. Behavioral specifications for specific objects are represented by predicates on the global history, restricted to an appropriate subset of possible communication events.

The $\mathcal{S}_{test}$ strategy blocks further execution once the system attempts to reach an illegal state violating the predicate on the global history. To test a particular object o against a behavioral specification $\langle o, \alpha, \mathcal{T}_\alpha \rangle$, the testing predicate can be expressed as $P(h) = h/\alpha \in \mathcal{T}_\alpha$. For behavioral requirements given as a predicate $P : \mathsf{Seq}[\alpha] \to \mathsf{Bool}$, defined by a convergent set of equations, membership in the trace set is effectively computable by reducing $P(h/\alpha)$ for the current global history h.

The $\mathcal{S}_{test}$ strategy is implemented in Maude by extending the conditional *exec* rule with a branch which checks the given predicate between each rewrite step and blocks execution if the predicate is violated. A Maude function *CheckPredicate* : *Pred* $\times$ *MsgList* $\to$ *Bool* is used for this purpose. A predicate is specified using a constant H which acts as a placeholder for the actual communication history. At run-time *CheckPredicate* parses the predicate specification against the actual history, calls any auxiliary predicates, and returns a boolean value indicating whether the history after the rewrite step would be in compliance with the predicate or not. If the execution is blocked by the strategy, the recorded history provides an error trace for the system run, describing how the specification was violated.

Example. The acceptable behavior of a philosopher behaving according to the *EatingPhil* interface (Sect. 2.3) can be expressed by a Maude operator *AccBeh*:

eq *AccBeh(nil)* = *true*

eq *AccBeh(H ; MSG from X to Y)* = *P(H/X ; MSG from X to Y)*

where P is the specification predicate of the *EatingPhil* interface, and where the notation h/X abbreviates $h/INOUT_X$. Since P in the Maude specification is a *global* predicate that spans all objects, there is no need to pass the object identifier as a separate parameter to *AccBeh*. In addition, since *AccBeh* is checked for each input and output event incrementally, we do not need to use the guarantee and assumption parts defined in Sect. 2.2.

5 Simulation of Open Environments for Testing

An open environment can be *simulated* such that the behavior of abstract objects is exclusively defined by the behavioral interfaces. Interface assumptions on the observable behavior may be used to generate arbitrary environment behavior within the limits imposed by the assumption predicate.

5.1 Syntactic Simulation of Open Environments

At the object-level, a rewrite theory is used to syntactically simulate the unknown environment. In an open environment, objects may be created and destroyed dynamically during execution. To mimic the open environment, we define a term containing a set *absIDs* of (abstract) object identifiers representing objects which may currently interact with the system: $\langle E : Envir \mid absIDs : \_, sysIDs : \_, seed : \_ \rangle$. The set *absIDs* will be used to generate input messages to the objects of the system. System objects are represented as a set *sysIDs* of pairs $Obj \times Set[Mtds]$ which consist of object identifiers and sets of method names corresponding to the alphabets of the object's interfaces. The messages emitted by abstract objects are input to the real objects of the system. The *seed* attribute is used for message generation.

In order to produce arbitrary but syntactically correct input to the system from objects in the environment, we need to select an object o from *sysIDs* and produce a message to o (either calling a method available in the interface of o or replying to a call from o found in the history). For this purpose, we use a pseudo-random number generator [15] and let the function $next : Nat \rightarrow Nat$ produce new seed values for the environment. Let the function $genMsg : Obj \times Obj \times Set[Msg] \times Nat \rightarrow Msg$ generate a new message *msg* to an object o with alphabet α in the system from an object in the environment, such that $msg \in \alpha$. The rewrite rule for message generation is given by:

$$\textbf{rl}\ [\text{msg-gen}] : \langle E : Envir \mid absIDs : o_1\, A, sysIDs : (o_2, \alpha)\, C, seed : X \rangle \quad \longrightarrow$$
$$\langle E : Envir \mid absIDs : o_1\, A, sysIDs : (o_2, \alpha)\, C, seed : next(X) \rangle\ genMsg(o_1, o_2, \alpha, X)$$

5.2 Semantic Simulation of Open Environments for Testing

At the metalevel, a rewrite theory is used to semantically simulate the unknown environment. Minimal behavioral requirements for open environments are given by assumptions in the system interfaces. Define a metalevel strategy $\mathcal{S}_{restrict}$ which *restricts* a rewrite system to behave according to a predicate on observable behavior. This strategy is similar to $\mathcal{S}_{test}$, but where $\mathcal{S}_{test}$ halts the execution when the application of an enabled rule would violate the predicate, $\mathcal{S}_{restrict}$ tries another enabled rule from the *labels* list of the *MetaRep* object instead. Open environments do not terminate; if no rewrite rule is applicable to any position of *curTerm*, the strategy changes the seed value and retries the rules.

The abstract environment specification can now be used as a *testbed* for an actual programmed component (see Fig. 3). Let $\mathcal{R}_{\mathrm{I}}$ be an object-level set of rewrite

	Rule set:	Configuration:
Metalevel rewrite system:	$\mathcal{S}_{restrict}(P_1(h/\alpha_1))$ $\wedge\ \mathcal{S}_{test}(P_2(h/\alpha_2))$	$\overline{\mathcal{R}_1} \cup \overline{\mathcal{R}_2}, (\overline{C_1}\ \overline{C_2})$, $\langle History : h \rangle$
	$\downarrow$ Control	$\uparrow$ History logger
Object level rewrite system:	$\mathcal{R}_1 \cup \mathcal{R}_2$	$C_1\ C_2$

Figure 3. Reflective testing of observable behavior in open environments.

rules generating (and possibly garbage collecting) messages. Rules from $\mathcal{R}_1$ may be applied to a configuration C_1 consisting of an *Envir* object. Let $\mathcal{R}_2$ be the object-level set of rewrite rules applicable to the concrete objects in a configuration C_2, e.g., the given component, with synchronization constraints on the internal state. Let α_1 and α_2 be alphabets associated with the objects of C_1 and C_2, respectively, such that $\alpha_1 \subseteq \alpha_2$. Let P_1 and P_2 be predicates observationally specifying the environment and actual component, respectively. If several interfaces are considered, P_1 will be the conjunction of assumptions and P_2 the conjunction of guarantees, restricted to the relevant alphabets. The metalevel strategy $\mathcal{S}_{restrict}$ restricts rule application from $\mathcal{R}_1$ to acceptable environment behavior, providing an abstract, open environment which may behave in any way that does not violate the predicate P_1. We here combine two metalevel strategies which react differently to the violation of predicates: $\mathcal{S}_{restrict}$ will restrict rule application so that the communication history conforms to the predicate, and $\mathcal{S}_{test}$ will halt the execution and produce an error object if the predicate does not hold. By specifying one predicate that spans only messages from the objects of the component, and one that spans all objects, and executing the former with $\mathcal{S}_{test}$ and the latter with $\mathcal{S}_{restrict}$, we can test whether the programmed component executes correctly provided that the environment does so.

5.3 Execution of the Philosopher Example

This test scenario was implemented in Maude by defining a metalevel rewrite rule *exec-test* similar to the rule given in Fig. 2, which combines the $\mathcal{S}_{restrict}$ and $\mathcal{S}_{test}$ strategies described above. The metalevel specification was used to test the implementation of philosophers described in Sect. 3.2. The test configuration consisted of one concrete philosopher object, rules for a table object, and an environment of 4 abstract philosophers, simulated as described in Sect. 5.1. The rewrite rules for philosopher behavior (Fig. 1) were compared to the *Phil* interface specification (Sect. 2.3) using $\mathcal{S}_{test}$, whereas application of the *msg-gen* rule was restricted by the $\mathcal{S}_{restrict}$ strategy to conform to the assumption A_{Phil}.

When the number of applications of the *exec-test* rule of this non-terminating specification was limited to 5000, the result (after 53494167 rewrites) was a trace

of 355 messages involving the concrete object. We observe that if rules which violate the guarantee specification are introduced, the violation will be detected by the strategy. Furthermore, if the environment assumptions are broken (e.g., by replacing the assumption predicate with the vacuous assumption *true*), this will cause a violation of the guarantee specification that will also be detected.

6 Related and Future Work

We do not attempt to fully survey the extensive literature on monitoring and testing here. Many previous history-based [8,19,22] and automata-based [2,21,23] approaches require specific and deterministic test cases to be defined. In contrast, we use *random testing* and assume-guarantee specifications to capture open environments, where environment behavior is arbitrary within the bounds of an assumption predicate. Invariant-driven strategies for Maude similar to our $\mathcal{S}_{restrict}$ have recently been proposed in [9], but that paper considers predicates on states rather than observable behavior and does not consider the application to open environments nor to testing. For open environments random testing within the bounds of minimal assumptions seems more attractive than deterministic tests.

The specifications of observable behavior considered in this paper are fairly easy to implement in rewriting logic. The specification language considered may be replaced by a more expressive language. For example, it would be interesting to combine our approach to open environment modeling with linear time temporal logic specifications on finite traces. An efficient algorithm in rewriting logic for the verification of such formulas has been given in [20].

7 Conclusion

The main contribution of this paper is to sketch an approach to the validation of black-box components in open environments by extending Maude models with a notion of observable behavior and related execution strategies. The paper shows how abstract specifications of open environments may be captured very naturally in a rewriting logic model extended with behavioral interfaces. The behavioral interfaces express safety requirements on the observable behavior of components. The approach is presented within a method-based, object-oriented setting, but may easily be adjusted to general asynchronous message passing. Due to the reflective character of rewriting logic, supported by Maude, it is possible to define execution strategies at the metalevel. In this paper, we have used this facility in four ways. First, a strategy is defined to non-deterministically generate arbitrary input to a system. Second, a strategy is defined to transparently introduce monitoring of a set of communication events. Third, a strategy is defined to restrict system input by semantic requirements on the observable behavior. Combining these strategies, the arbitrary behavior of open environments may be simulated within the bounds of minimal assumptions. The separation of object-level and metalevel constraints facilitates experimenting with different assumptions on the environment. The same

approach may also be used to execute a prototype model defined by its observable behavior, before deciding on its implementation details. Fourth, a strategy is defined to test whether an executable model is well behaved with respect to semantic requirements on the observable behavior. Combining all four strategies, we obtain abstract validation environments for models of components or distributed applications, in which the environment is unspecified but subjected to minimal observational requirements.

Acknowledgments. We are grateful to Eyvind W. Axelsen for contributing to the implementation of these ideas and to the anonymous referees for helpful comments.

References

[1] Alpern, B. and F. B. Schneider, *Defining liveness*, Information Processing Letters **21** (1985), pp. 181–185.

[2] Barbey, S., D. Buchs and C. Péraire, *A theory of specification-based testing for object-oriented software*, in: *Proc. Eur. Dependable Computing Conf. (EDCC2)*, LNCS **1150** (1996), pp. 303–320.

[3] Clavel, M., "Reflection in Rewriting Logic: Metalogical Foundations and Metaprogramming Applications," CSLI Publications, Stanford, California, 2000.

[4] Clavel, M., F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer and J. F. Quesada, *Maude: Specification and programming in rewriting logic*, Theoretical Computer Science **285** (2002), pp. 187–243.

[5] Clavel, M. and J. Meseguer, *Reflection in conditional rewriting logic*, Theoretical Computer Science **285** (2002), pp. 245–288.

[6] Dahl, O.-J., *Can program proving be made practical?*, in: M. Amirchahy and D. Néel, editors, *Les Fondements de la Programmation*, Institut de Recherche d'Informatique et d'Automatique, Toulouse, France, 1977 pp. 57–114.

[7] Dahl, O.-J., "Verifiable Programming," Prentice Hall, New York, N.Y., 1992.

[8] Doong, R.-K. and P. G. Frankl, *The ASTOOT approach to testing object-oriented programs*, ACM Trans. Softw. Eng. Methodol. **3** (1994), pp. 101–130.

[9] Durán, F., M. Roldán and A. Vallecillo, *Invariant-driven strategies for Maude*, ENTCS **124** (2005), pp. 17–28, Proc. 4th Intl. Workshop on Reduction Strategies in Rewriting and Programming (WRS 2004).

[10] Goguen, J. A., T. Winkler, J. Meseguer, K. Futatsugi and J.-P. Jouannaud, *Introducing OBJ*, in: J. A. Goguen and G. Malcolm, editors, *Software Engineering with OBJ: Algebraic Specification in Action*, Kluwer, 2000 pp. 3–167.

[11] Hoare, C. A. R., "Communicating Sequential Processes," Prentice Hall, 1985.

[12] International Telecommunication Union, *Open Distributed Processing - Reference Model parts 1–4*, Technical report, ISO/IEC, Geneva (1995).

[13] Johnsen, E. B. and O. Owe, *A compositional formalism for object viewpoints*, in: B. Jacobs and A. Rensink, editors, *Proc. 5th Intl. Conf. on Formal Methods for Open Object-Based Distributed Systems (FMOODS'02)* (2002), pp. 45–60.

[14] Johnsen, E. B. and O. Owe, *Object-oriented specification and open distributed systems*, in: O. Owe, S. Krogdahl and T. Lyche, editors, *From Object-Orientation to Formal Methods: Essays in Memory of Ole-Johan Dahl*, LNCS **2635**, Springer-Verlag, 2004 pp. 137–164.

[15] Johnsen, E. B., O. Owe and E. W. Axelsen, *A run-time environment for concurrent objects with asynchronous method calls*, in: N. Martí-Oliet, editor, *5th Intl. Workshop on Rewriting Logic and its Applications (WRLA'04)*, ENTCS **117** (2005), pp. 375–392.

[16] Jones, C. B., "Development Methods for Computer Programmes Including a Notion of Interference," Ph.D. thesis, Oxford University, UK (l981).

[17] Meseguer, J., *Conditional rewriting logic as a unified model of concurrency*, Theoretical Computer Science **96** (1992), pp. 73–155.

[18] Meseguer, J. and G. Roşu, *Rewriting logic semantics: From language specifications to formal analysis tools*, in: D. A. Basin and M. Rusinowitch, editors, *Proc. 2nd Intl. Joint Conf. on Automated Reasoning (IJCAR 2004)*, LNCS **3097** (2004), pp. 1–44.

[19] Pretschner, A., H. Lötzbeyer and J. Philipps, *Model based testing in incremental system development*, Journal of Systems and Software **70** (2004), pp. 315–329.

[20] Roşu, G. and K. Havelund, *Rewriting-based techniques for runtime verification*, Journal of Automated Software Engineering **12** (2005), pp. 151–197.

[21] Rusu, V., H. Marchand, V. Tschaen, T. Jéron and B. Jeannet, *From safety verification to safety testing*, in: R. Groz and R. M. Hierons, editors, *16th IFIP Intl. Conf. on Testing of Communicating Systems (TestCom 2004)*, LNCS **2978** (2004), pp. 160–176.

[22] Tyler, B. and N. Soundarajan, *Black-box testing of grey-box behavior*, in: A. Petrenko and A. Ulrich, editors, *3rd Intl. Workshop on Formal Approaches to Testing of Software (FATES 2003)*, LNCS **2931** (2004), pp. 1–14.

[23] Van Aertryck, L., M. Benveniste and D. Le Metayer, *CASTING : A formally based software test generation method*, in: *Intl. Conf. on Formal Engineering Methods (ICFEM'97)* (1997), pp. 101–110.

[24] Venkatasubramanian, N. and C. L. Talcott, *Reasoning about meta level activities in open distributed systems*, in: *Proc. 14th Symp. on Principles of Distributed Computing (PODC '95)* (1995), pp. 144–152.

[25] Walter, T., I. Schieferdecker and J. Grabowski, *Test Architectures for Distributed Systems - State of the Art and Beyond*, in: A. Petrenko and N. Yevtushenko, editors, *11th Intl. Workshop on Testing Communicating Systems (IWTCS)*, IFIP Conference Proceedings **131** (1998), pp. 149–174.

Chapter 9

Paper #2: Validating behavioral component interfaces in rewriting logic

Einar Broch Johnsen, Olaf Owe and Arild B. Torjusen

Published in *Fundamenta Informaticae*, 82(4), 2008 [JOT08].

Chapter 10

Paper #3: Towards integration of XML in the Creol object-oriented language

Arild Torjusen, Olaf Owe and Gerardo Schneider

Published as Research Report 365, Dept. of Informatics, Univ. of Oslo, Oct. 2007 (revised Feb. 2008) [TOS07a].

Universitetet i Oslo
Institutt for informatikk

Towards integration of XML in the Creol object-oriented language

Arild Torjusen, Olaf Owe, and Gerardo Schneider

Research report 365
ISBN 82-7368-323-0
(Revised version)

October 2007
Revised February 2008

Towards integration of XML in the Creol object-oriented language

Arild Torjusen, Olaf Owe, and Gerardo Schneider

Department of Informatics, University of Oslo
PO Box 1080 Blindern, NO-0316 Oslo, Norway
email: {aribraat,olaf,gerardo}@ifi.uio.no

Abstract

The integration of XML documents in object-oriented programming languages is becoming paramount with the advent of the use of Internet in new applications like web services. Such integration is not easy in general and demands a careful language design. In this paper we propose an extension to Creol, a high level object-oriented modeling language for distributed systems, for handling XML documents.

1 Introduction

XML (eXtensible Markup Language) [7] is a flexible and generic format for structured data aimed at being shared on the World Wide Web and intranets. The need for XML documents as first-class citizens is acknowledged by academic as well as by business-oriented communities [21].

XML *documents* are ordered labeled tree structures containing *markup* symbols describing their content. The document structure is described by a document type -or *schema*- written in a schema language. Many such languages have been proposed, among them DTD (Document Type Definition) [7] and XML-Schema [10]. Unlike other markup languages (like HTML), XML has no restrictions on the tags or attributes used to mark up a document. One remarkable feature of XML is its plain-text-based nature. The advantage is that there is no problem with proprietary nor deciphering data. The disadvantages are the large bandwidth needed for transmission of documents and the need of encryption because of security issues. Part of the manipulation of XML documents includes the retrieval of information through queries. XQuery [5] provides a sound foundation for XML query, based on infosets. The situation is not ideal for developers since they need to know one language for analyzing the tuples e.g., SQL, another language for the Infoset e.g., XQuery, and a third one for operating on objects e.g., Java. Some attempts have been done to combine object-oriented languages and XML, but this turned out to be a complex task; this problem is known as the *impedance mismatch* [25], which arises when trying to combine object-oriented programming languages and (relational) databases.

The integration of XML on current object-oriented languages is far from trivial. The initial approach has been to treat XML through APIs which uses strings for representing literals. One problem of this approach is that it limits the use of static checking tools. Furthermore, the representation of programs as text involves potential security risks. See [21] for a more detailed description of the main problems arising with the integration of XML in object-oriented languages.

In addition to the integration of XML documents within OOP languages, another question is what to do with these data, i.e., how easy it is to make queries, getting useful information from such XML-documents.

1.1 Creol

Our research project concerns integration of XML into the object-oriented language Creol [16, 15, 18]. The main features of Creol are:

- It supports both object-oriented classes, with late binding and multiple inheritance, as well as user defined data types and functions. This gives flexibility in our choices when representing XML.
- It is oriented towards open distributed systems. Exchange of XML documents fits naturally in this context.
- It supports concurrency and method calls based on asynchronous communication. We wish to explore the processing and sharing of XML documents in this setting.
- It is strongly typed, supporting subtypes and subinterfaces, with a type hierarchy including both by means of the universal type, *Data*.
- It has a formal operational semantics, defined in rewriting logics. This enables us to formalize the extension to XML by reuse of the operational semantics.
- It has a small kernel with an operational semantics consisting of only 11 rewrite rules. This makes it easy to extend and modify the language and the semantics.
- Creol has an executable interpreter defined in the Maude language. This provides a useful framework for implementation and testing of our XML representations.

1.2 Related Work

The list of languages for processing XML documents is extensive, so it is not possible to be exhaustive here. We briefly discuss below some of the most influential works, namely XDuce, CDuce and C_ω. We mention other related work as reference for further reading, without entering into detail.

XDuce XDuce [13] is a functional programming language for XML processing. Its basic data values are XML documents and its types—called *regular expressions types*—correspond to document schemas. The language is statically typed but it also provides dynamic type-checking. Other interesting feature of XDuce is regular expression pattern matching which includes tag checking, subtree extraction and conditional branching.

An XML document in XDuce is represented as a sequence of nodes, and types use similar constructs as string regular expressions like "*" for representing that zero or more occurrences may happen, "?" for indicating an item may be omitted, "+" for one or more time repetition, "|" for alternation and "," for concatenation. The main difference with string regular expressions, is that regular expression types describe sequences of tree nodes instead of sequences of characters.

The type-checking algorithm is based on the following subtype relationship: one type is a subtype of another if and only if the former denotes a subset of the latter. The subtype checker may be used both for checking that the actual type of a function's body is a subtype of the programmer-declared result type and for verifying function call arguments against parameter types given by the programmer. Although the theoretical complexity of the corresponding problem to subtype checking on tree automata is exponential, it is claimed in [13] that it works well in practice.

CDuce CDuce [3] is a typed functional language born from an attempt to solve some of the limitations of XDuce [13]. It extends XDuce on three areas:

Type system In addition to regular expression types and type-based patterns, CDuce adds recursive types and other less XML specific constructs: products, records (open and closed), general Boolean connectives (intersection, union and difference) and arrow types. This extension takes care of not breaking down the nice subtype relation of XDuce.

Language design The following language constructions are included in CDuce: overloaded functions (useful for code sharing and reuse), iterators on sequences and trees and other extensions of the pattern algebra. Besides, XML tags are first-class citizens and strings are simple sequence of characters. The language support higher-order programming, so all functions are first-class citizens.

Run-time system A new approach for avoiding unnecessary computation at runtime is added in CDuce, allowing the programmer to use a more declarative style when writing patterns, without degrading performance. The underlying theory is based on a new kind of tree automata.

CDuce provides also a tool for translating DTDs into CDuce's types.

C$_\omega$ C$_\omega$ [24] is a programming language developed at Microsoft Research, combining features from two other research languages: (a) Polyphonic C# [2]: a control flow extension with asynchronous wide-area concurrency, and (b) Xen [21]: a data type extension for processing XML and table manipulation. Besides other interesting features, C$_\omega$ allows the construction of objects using XML syntax.

The C$_\omega$ type systems combines the following three data models: relational, object and XML data-access, and it is more oriented to XML constrained using W3C XML Schema. The language covers the following XML and XML Schema features: document order, distinction between elements and attributes, multiplicity of fields with equal name but different values and content models for specifying choice (union) types for fields.

One of the nice features of the C$_\omega$ type system are *streams*. It is possible to invoke methods on streams, which are applied to all the elements of the stream; XPath-style queries over objects graphs are easily written in this way. It also includes the concept of *apply-to-all* expressions construct. Choice (union) types allow the programmer to specify one of different possible values for a certain field. Moreover, *null* is a valid value for a type, which have been proved useful in XML and relational databases. Document order and multiplicity of equal names for child elements, are solved through the use of *anonymous structs*. In C$_\omega$ DTDs (and XML Schemas) are represented by *content classes*.

Other languages The following languages try to extend Java with XML processing: XJ [12], XACT [20], XOBE [19], BPELJ [4].

XL [11] is a language whose only type system is the XML type system, and not a language whose syntax is described using XML vocabulary. It is specially designed for the implementation of Web services. XL is portable and fully compliant with all W3C standards such as XQuery, XML Protocol, and XML Schema.

PiDuce[1] is CDuce-like language based on the π-calculus. ECMAScript for XML (E4X) is a set of programming language extensions adding native XML support to ECMAScript. E4X is standardized by Ecma International in ECMA-357 standard.[2]

See [22] for a good survey on static type-checking for XML transformation languages.

1.3 Our Agenda

In order to integrate XML documents in Creol, we intend to follow the following agenda:

1. *Parsing and well-formedness checking.* We will enhance the language as to be able to take a given XML document as input and generate some internal data structure from it.

2. *Internal representation of XML in Creol.* We aim at extending Creol for supporting XML documents with the least possible changes to the existing framework. One of the key features we would like to preserve is Creol static type-safety. In order to make a lightweight integration of XML into Creol and keep static type safety we will restrict type checking of XML in this implementation to only *well-formedness* of XML values, i.e. that some value of type `XMLDoc` (the Creol type for XML documents) checks out as an `XMLDoc`.

3. *Simple validity-checking of XML data-structures.* We will validate XML data-structures against some schema. Schema is here taken in a broad sense, meaning a formal description of the type of an XML document, without regards to any specific schema language as e.g. DTD, XML-Schema or RELAX NG (cf. Sec 3). Validity checking will be done by functions "on top" of the type system and not within the type system itself.

4. *More complex validity-checking of XML data-structures.* We will perform more complex validity checking after enhancing the Creol language with *regular expression types*, following the work of Hosoya et.al. [14].

5. *Queries.* We will also demonstrate how to perform queries and data extraction from XML document instances.[3]

6. *Transformations.* We will perform more complex operations such as construction and transformations on XML documents.

In this paper, however, we will concentrate on items 2 and 3 above. In the next section we show how XML documents are integrated in Creol. In Section 3 we show how schemas are represented in Creol after a short discussion on existing schema languages. Section 4 is concerned with the validation of XML documents. In Section 6 we conclude and present further work.

[1] `http://www.cs.unibo.it/~laneve/PiDuce/`

[2] See `http://www.ecma-international.org/publications/standards/Ecma-357.htm`.

[3] Cf. e.g. `http://www.w3.org/TR/2005/WD-xquery-use-cases-20050915/` for test use cases.

2 A model for XML in Creol

Different XML documents may vary in physical representation due to syntactic changes permitted by the XML standard. W3C has issued a recommendation which describes how any XML document can be normalized into a canonical form [6]. The data model defined in the XPath 1.0 Recommendation [26] is the basis for canonical XML and we will use this as the point of departure for the internal representation of XML in Creol.

2.1 The XPath Data model

XPath models an XML document as an ordered tree containing nodes of seven different types:

- root: The root node is the root of the tree and will correspond to an XML document instance. It contains a list of processing instructions, a list of comment nodes, and exactly one element which is the root element of the document.
- element: The element node has a name (corresponding to the XML tag for the element) and may have as its children element nodes, comment nodes, processing instruction (PI) nodes and text nodes. It is also associated to a set of attribute nodes and a set of namespace nodes.
- text: A text node contains a string, representing character data in the XML document.
- attribute: An attribute node contains a name and a value.
- namespace: A namespace node contains a string value for the namespace prefix and a value for the namespace URI.
- processing instructions: A PI node has a name identifying the target application and a string which is to be passed to the application.
- comment: A comment node contains a string.

To simplify the initial XML implementation for Creol we will leave out the last three kinds of nodes from our model. According to [7], comments "are not part of the document's character data; an XML processor MAY, but need not, make it possible for an application to retrieve the text of comments.", we choose not to retain comments in the Creol representation of XML. Processing instructions are not relevant for our purpose of demonstrating lightweight integration of XML in Creol and can also be left out. As will be explained later we will adopt the DTD language for specification of schemas; since the DTD does not support namespaces it is natural not to represent namespace nodes in the model. These design choices also simplifies the definition of element and root nodes.

2.2 The Creol representation of XML

Given the two-tiered type-system of Creol where objects are typed by interfaces and local computations on terms occur in a functional language, we introduce XML into Creol by adding type constructors for a new `XMLDoc` type, as a subtype of the universal type `Data`, as well as functions on this type.

Creol has an operational semantics defined in rewriting logic, which is executable with Maude [9] and provides an interpreter and analysis platform for system models. So to accommodate XML we extend the operational semantics with some Maude sorts (type names) and constructors (Creol definitions would be very similar):

```
sorts   XMLName ElemNd TextNd AttNd ContentNd XMLDoc .
subsort ElemNd TextNd < ContentNd .
subsort String < XMLName .
```

introducing sorts for XML names, element, attribute, text and content nodes, letting `ElemNd` and `TextNd` be subsorts of `ContentNd`. The sort `XMLName` includes `String`, the predefined sort for strings.

To simplify the writing of XML values in a program we use mix-fix notation (indicating argument positions by underline symbols) to provide a compact syntax by adding the following constructors for attributes, text nodes and elements (with and without attributes).

```
op (_=_)   : XMLName String                  -> AttNd  [ctor] .
op _(_)[_] : XMLName AttNdList ContentNdList -> ElemNd [ctor] .
op _[_]    : XMLName ContentNdList           -> ElemNd [ctor] .
op tx      : String                          -> TextNd [ctor] .
```

where the clause [ctor] after an operator (op) indicates that it is a constructor, and where `ContentNdList` and `AttNdList` represent lists of `ContentNd` and `AttNd`, respectively, defined as conventional in Maude.

Note that there is no specific constructor for root nodes. Since we leave out processing instructions and comments, the root node is just the element node occurring at the root of an XML document tree. Thus, the XML document constructor is

```
op xmlDoc   : ElemNd XMLSchema -> XMLDoc [ctor] .
```

We define the operator

```
op noSchema :                  -> XMLSchema [ctor] .
```

for XML documents with no XMLSchema. Other XMLSchema constructors are defined further below.

Example The following simple XML fragment

```
<email>
  <head>
    <sender>Arild</sender>
    <rcp addr="vera@foo.com">Vera</rcp>
    <subject>Test</subject></head>
  <body>
    <message>Hello there, you wrote in an earlier message:
    <quote>We'll meet again</quote> See you later</message>
  </body>
</email>
```

can be represented as an `ElemNd` term with the Creol/Maude syntax:

```
"email"[
  ("head"[
    ("sender"[tx("Arild")])
    ("rcp"("addr"="vera@foo.com")[tx("Vera")])
    ("subject"[tx("Test")])])
  ("body"[
    ("message"[
    tx("Hello there, you wrote in an earlier message:")
    ("quote"[tx("We'll meet again")]) tx("See you later")])])] .
```

As conventional in Maude, the list constructor (concatenation) is here denoted by white space (blank).

3 Schemas and type checking

3.1 Regular expression types vs. schema types

Static type checking of XML documents in a programming language can be achieved by introducing types for XML fragments in the language. Xduce and CDuce mentioned earlier are examples of projects going in this direction, by introducing regular expressions types for XML schemas and letting the type system handle the validation.

For the current integration of XML in Creol we will take a less involved approach by introducing one data type for XML schema, together with functions to validate documents against schema. We may then specify a type for an XML document as a value of type `XMLSchema` and thus the validation takes place within the existing type system and does not constitute an addition to the type system itself. The advantage of this approach is that we do not need significant modifications to the type system, the disadvantage is that we get a less fine grained tool for working with XML schema.

3.2 Expressive power of schema languages

There exists several generally adopted XML schema languages with different expressive power. Murata, Lee, and Mani [23] suggest a taxonomy of schema languages based on the formal theory of regular tree grammars. Some of the most common schema languages can be ranked in order of increasing expressivity thus: The DTD language, The W3Cs XML Schema, The RELAX NG specification. Validation of the first two can be done by simple adaptions of word automata, while the last requires a more complicated tree automaton. However the DTD language is sufficiently expressive for our purpose which is to demonstrate how XML can be integrated in the object oriented modeling framework of Creol. Therefore in our model for XML schema values in Creol we adapt the restrictions inherent in the DTD language to achieve simple validation, (i.e. only deterministic regular expressions is allowed in the definition of an element as explained below).[4]

[4]Roughly corresponding to "Local Tree Grammars" in [23].

3.3 The schema type for Creol

A DTD is a list of markup declarations where markup declarations are either element type declarations, attribute-list declarations, entity declarations, or notation declarations.

For our purpose we only consider element type declarations and attribute-list declarations. Entity declarations may be considered as a kind of macro notation for strings that may appear in a DTD or an XML document, since our focus is on internal processing we will assume that these already are expanded by the parser and will abstract away from them in our model. Notation declarations are similarly a kind of shorthand for notations and are also left out. Accordingly the XML Schema constructor is:

```
op xmlSchema : XMLName ElemDeclList AttDeclList -> XMLSchema .
```

Element type declarations consist of a name referring to an element and a specification of the legal content. There are four kinds of specifications: either one of the designated keywords "EMPTY" or "ANY", or the specification of a *content model*. A content model is a context free grammar governing the allowed types of the child elements and the order in which they are allowed to appear. The fourth kind of content specification is the *Mixed-content Declaration* which is of the form:

$$(\,\texttt{\#PCDATA} \,|\, e_1 \,|\, e_2 \,|\, \ldots \,|\, e_n\,)*$$

Where each e_i is an element name and n may be 0 in which case the '*' is optional.

Example A DTD for the XML fragment given above could be:

```
<!ELEMENT email (head, body, foot*) >
<!ELEMENT head (sender, rcp, subject?)>
<!ELEMENT body (message)*>
<!ELEMENT foot (#PCDATA)>
<!ELEMENT sender (#PCDATA)>
<!ELEMENT rcp (#PCDATA)>
<!ELEMENT subject (#PCDATA)>
<!ELEMENT message (#PCDATA|quote)*>
<!ELEMENT quote (#PCDATA)>
```

The first three element declarations specify content models and the rest are instances of mixed-content declarations. We model the content models as *regular expressions*. Let Σ be an alphabet over element names, including the reserved name `PCDATA`. By including `PCDATA` in Σ we can model a mixed-content declaration as a special kind of a content model specification. The set of regular expressions over Σ^* are obtained in the standard way: The empty string ϵ and each member of Σ are regular expressions. If α is a regular expression, then so are (α), $\alpha?$, $\alpha*$ and $\alpha+$. If α and β are regular expressions, then so is $\alpha\ \beta$, and $\alpha \,|\, \beta$. The operators $?$, $*$, and $+$ has higher precedence than concatenation. Concatenation has higher precedence than union (|). The regular expression combinators have the expected semantics. We model element declarations as follows:

```
subsort XMLName < ReToken < RegExp .
op elemDecl    : XMLName ContentModel -> ElemDecl [ctor] .
op empty       :                      -> ContentModel [ctor]  .
```

```
op any          :                       -> ContentModel [ctor]  .
op elemCt       : RegExp                -> ContentModel [ctor] .
op PCDATA       :                       -> ReToken .
op _?           : RegExp                -> RegExp [ctor prec 40] .
op _*           : RegExp                -> RegExp [ctor prec 40] .
op _+           : RegExp                -> RegExp [ctor prec 40] .
op _@_          : RegExp RegExp         -> RegExp [ctor assoc prec 42][5]
op _|_          : RegExp RegExp         -> RegExp [ctor prec 44]
```

The XML specification adds the requirement that the content models must be deterministic [7, Appendix E], i.e. a content model must not allow an element to match more than one occurrence of an element name in the content model. This ensures that when matching an element name σ with a schema we do not have to look ahead beyond the σ in the input string to decide which regular expression in the content model matches σ. This requirement is included in the XML specification to ensure compatibility with SGML. For a detailed discussion see e.g. [8].

Example The Maude syntax for a document type declaration containing the DTD given above is:

```
xmlSchema("email",(
elemDecl("email", elemCt("head"@"body"@("foot"*)))
elemDecl("head", elemCt("sender"@"rcp"@("subject"?)))
elemDecl("body", elemCt("message"*))
elemDecl("foot", elemCt(PCDATA))
elemDecl("sender" ,elemCt(PCDATA))
elemDecl("rcp", elemCt(PCDATA))
elemDecl("subject" ,elemCt(PCDATA))
elemDecl("message", elemCt((PCDATA|"quote")*))
elemDecl("quote", elemCt(PCDATA)), noAttDecl[6]) .
```

4 Validating XML in Creol

Well-formedness of any value of type `XMLDoc` is ensured by Maude type checking. The XML specification defines an XML document to be *valid* "if it has an associated document type declaration and if the document complies with the constraints expressed in it" [7].

The XML document constructor associates the root element of a document with a schema, (which may also be the special value `noSchema`). Hence, an XML document is validated by first checking for existence of a schema and by checking that the root node element name matches that schema name. Secondly we check that each element node in the tree is valid with respect to the element declarations in the schema.

Validation of a document is performed by the function

```
op validate : XMLDoc -> ValResult .
```

[5] We here use '`@`' as the concatenation operator to avoid problems with overloading of ',' or whitespace.
[6] Attribute declarations are not yet supported.

```
op res : Bool String -> ValResult .
eq collate( res(b,s) , res(b',s')) = res((b and b') , (s + s')) .

eq validate( xmlDoc(nm(atts)[cts], noSchema )) = res(false,"No Schema") .

eq validate( xmlDoc(nm(atts)[cts] , xmlSchema(nm',elDs,attDs) ) ) =
  if ( nm =/= nm' ) then res(false,("Document root-element: " + nm +
                ", must match schema type: " + nm' + " \n"))
  else val(nm(atts)[cts] , elDs) fi .

eq val(tx(str),elDs) = res(true,"") .
eq val(emp,elDs) = res(true,"") .
eq val((ct cts), elDs) = collate( val(ct,elDs) , val(cts,elDs)) [owise] .

ceq val(nm(atts)[cts],elDs) =
  if cm == undefined then
    res(false,("Element-type :" + nm + " must be declared.\n"))
  else  check(nm(atts)[cts],cm,elDs) fi     if cm :=  getCM(nm,elDs) .

eq check(nm(atts)[cts],empty,elDs) =
  if (cts == emp) then res(true,"Empty elem: " + nm + "\n")
  else
    res(false,"Elem: " + nm + " declared as EMPTY, but has content.\n")
  fi .

eq check(nm(atts)[cts],any,elDs) =
  collate(res(true,"Elem: " + nm + " defined as ANY.\n"),val(cts,elDs)) .

eq check(nm(atts)[cts], elemCt(regexp) ,elDs) =
  if match(getTokens(cts), regexp) then
    collate (res(true, nm + ": (" + ctToS(cts) + ")
            matches [" + reToS(regexp) + "]\n") , val(cts,elDs))
  else
    collate (res(false, nm + ": (" + ctToS(cts) + ")
            does NOT match [" + reToS(regexp) + "]\n"), val(cts,elDs) ) fi .
```

Figure 1: Maude code for validation of XML documents.

where a `ValResult` is a boolean/string pair with the boolean value indicating validity and the string containing an error message or a record of the processing. `validate` checks whether there is a schema with a name matching the document root node associated with the document, in which case the recursive function `val` is called, otherwise validation ends with a negative result. The helper function `collate` builds the final validation result for a document from validation of its parts. The relevant parts of the Maude code are given in fig. 1. The function:

```
op val : ContentNdList ElemDeclList -> ValResult .
```

validates a content node list against the element declaration list defined by the schema. For a list of nodes, `val` is called recursively on each node in the list. For a single node, the element

type declaration corresponding to the node is retrieved (by name) from the list of element declarations and the node is checked against the retrieved declaration by a call to the function

```
op check : ContentNd ContentModel ElemDeclList -> ValResult .
```

If there is no `ContentModel` for some node the document is invalid. Note also that according to [7] an element type must not be declared more than once so uniqueness of declarations may be assumed.

In the call to `check`, the complete list of element declarations is passed on as a parameter since any child nodes to the node currently being processed must also be validated.

For a `ContentNd` to be valid relative to a `ContentModel` we need to consider three cases: The `ContentModel` is `empty` and the element should have no content or the `ContentModel` is `any` and the element can consist of any sequence of (declared) elements intermixed with character data. These two cases are easy to check. The third case is where the `ContentModel` specifies a regular expression, in this case the function `match` will be called to determine whether the list of actual children elements matches the regular expression specified in the corresponding element declaration, in addition `val` is called on the list of children elements.

The function

```
op getTokens : ContentNdList -> TokenList .
```

builds a list of tokens from the element content, i.e. it builds a list consisting of; element names for content nodes of sort `ElemNd`, and the special token `'PCDATA` for content nodes of sort `TextNd`. As tokens we use the Maude built-in sort `Qid`. The token list and the regular expression from the element type declaration are then processed by the `match` function:

```
op match : TokenList RegExp -> Bool .
```

Matching of a list of element names from Σ against a regular expression is implemented by constructing a deterministic finite automaton from the regular expression and test whether the automaton accepts the string corresponding to the list of names. The implementation details are left out here, but see e.g. [1,17] for a description of how this is done in Maude. Our implementation is based on the work done in [1]. `ctToS` and `reToS` are just string conversion functions for content nodes and regular expressions for logging purposes.

Example Validation of the sample document with the DTD specified above gives the following result:

```
reduce in XML-VALIDATE-TEST : validate(xmlDoc(email, emailSchema)) .
rewrites: 9454 in 8ms cpu (8ms real) (1050561 rewrites/second)
result ValResult:
res(true, "email: (head ,body) matches [head @ body @ (foot*)]
           head: (sender ,rcp ,subject) matches [sender @ rcp @ (subject?)]
           sender: (PCDATA) matches [PCDATA]
           rcp: (PCDATA) matches [PCDATA]
           subject: (PCDATA) matches [PCDATA]
           body: (message) matches [(message*)]
           message: (PCDATA , quote ,PCDATA) matches [(PCDATA | quote*)]
           quote: (PCDATA) matches [PCDATA]")
```

5 A Creol example

The current implementation of XML in Creol and the validation algorithm enable Creol applications to use XML documents as a data storage and exchange format.

As an example we look at a simple system consisting of a LibraryServer and a LibraryClient which exchange messages on XML format. The example shows how XML elements and XML documents can be used as parameters in method calls and returns and it also illustrates validation of the XML Data against DTDs, the Creol code is given in fig. 2. The library server keeps a catalogue of books in an XML document and a client may call the method

getEntries(**in** query:ElemNd ; **out** result:ElemNd)

```
class LibraryClient implements LibraryCl
begin
  var res1 : ElemNd var res2 : ElemNd
  var res3 : ElemNd var res4 : ElemNd

  op run == var server : LibraryServ ;
    server := new LibraryServer();
    server.getEntries("query"[("title"[tx("TCP/IP Illustrated")])] ; res1) ;
    server.getEntries("query"[("price"[tx("65.90")])] ; res2) ;
    server.getEntries("query"[("publisher"[tx("Addison−Wesley")])];res3) ;
    // Initialize a new server with a invalid catalogue.
    server := new LibraryServer(xmlDoc("bib"[tx("A non valid library catalogue")],noSchema)) ;
    server.getEntries("query"[("title"[tx("TCP/IP Illustrated")])] ; res4)
end

class LibraryServer(catalogue:XMLDoc) implements LibraryServ
begin
  var queryType : XMLSchema := <queryTypeValue>
  var defCat : XMLDoc := <catalogueValue>
  var status : String

  //Use a default catalogue if no parameter is given to class.
  op init == if catalogue = null then catalogue := defCat end

  with LibraryCl
  op getEntries(in query:ElemNd ; out result:ElemNd ) ==
    //Make sure that library catalogue is valid before accepting calls.
    if xmlValid(catalogue) then
      status := "Valid, accepting calls";
      if xmlValid(query , queryType) then
        result := subElemQuery(query,catalogue)
      else
        result := "result"[("err_result"[tx("Invalid query type")])]
      end
    else
      status := "Invalid catalogue";
      result := "result"[("err_result"[tx("Library catalogue invalid")])]
    end
end
```

Figure 2: Creol program

on the server. The server will first ensure that its own catalogue is valid w.r.t. a DTD for the library catalogue with a call to the function

xmlValid(catalogue:XMLDoc) .

The server then checks that the query conforms to the specified DTD for queries with a call to the function

xmlValid(query:ElemNd,queryType:XMLSchema),

and executes the query by calling the function

subElemQuery(query:ElemNd,catalogue:XMLDoc) .

If the query is valid, the server will perform the query given as a parameter and return an `ElemNd` as response, if it is not valid it will return an `ElemNd` containing an error message as a response, the same will happen if the catalogue is invalid. All responses conform to a query result DTD.

To execute the program we extend the Creol language with the three functions mentioned above. For the two validation functions this amounts to extending the Maude interpreter by specifying equations which map the Creol syntax above to our previously defined `validate` function in Maude to enable the interpreter to execute the program as a Creol program. With `D` and `D'` being of sort `Data` we add the following two equations to the interpreter:

```
eq "xmlValid"(D)  = bool(getB(validate(D))) .
eq "xmlValid"(D # D')  = bool(getB(validate(xmlDoc(D,D')))) .
```

The subElemQuery function is another addition to the Creol API and is likewise implemented in Maude. We leave out the details of the implementation since the function is tailored for this specific case for the purpose of the example. To extend the Creol API with more general XML operations we would need to consider carefully which operations to add to the API to ensure that we chose a minimal set of useful basic functions. This is left for further work.

In the program text above, the <queryTypeValue> is a Creol value representing the DTD for queries. the <catalogueValue> is a Creol value representing an XML document instance. See the appendix for an example of an execution of the program with some specific values.

6 Conclusion

Integrating XML documents in object-oriented languages is not easy in general as witnessed by the extensive research conducted in this area, and nicely presented in the survey [21]. We have shown here how to integrate XML documents into Creol, an object-oriented language with formal semantics in rewriting logic. We have also presented an algorithm for validating XML documents against XML schemas, to show that the former are instances of the latter.

This paper is a first step towards a full integration of XML into Creol, and we intend to pursue our work as to complete our agenda described in Section 1.3. In particular, we find it extremely interesting to be able to manipulate and reason about XML documents, to include regular expression types, and to adapt the semantic sub-typing algorithm from CDuce and XDuce discussed in the introduction.

References

[1] E. W. Axelsen. A meta-level framework for recording and utilizing communication histories in Maude. Master's thesis, Dept. of Informatics, Univ. of Oslo, Norway, Aug. 2004.

[2] N. Benton, L. Cardelli, and C. Fournet. Modern concurrency abstractions for c#. *ACM Trans. Program. Lang. Syst.*, 26(5):769–804, 2004.

[3] V. Benzaken, G. Castagna, and A. Frisch. CDuce: an XML-centric general-purpose language. *SIGPLAN Not.*, 38(9):51–63, 2003.

[4] M. Blow, Y. Goland, M. Kloppmann, F. Leymann, G. Pfau, D. Roller, and M. Rowley. BPELJ: BPEL for java. http://ftpna2.bea.com/pub/downloads/ws-bpelj.pdf.

[5] S. Boag, D. Chamberlin, M. Fernández, D. Florescu, J. Robie, and J. Siméon. *XQuery 1.0: An XML Query language*, November 2005. http://www.w3.org/TR/2004/WD-xquery-20040723/.

[6] J. Boyer. *Canonical XML Version 1.0*. W3C, 2001. W3C Recommendation 15 March 2001. http://www.w3.org/TR/xml-c14n.

[7] T. Bray, J.Paoli, C. Sperberg-McQueen, E. Maler, and F. Yergeau. *Extensible Markup Language (XML) 1.0*, third edition, February 2004. http://www.w3.org/TR/REC-xml/.

[8] A. Brüggemann-Klein and D. Wood. Deterministic regular languages. In *STACS*, pages 173–184, 1992.

[9] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Maude: Specification and programming in rewriting logic. *Theoretical Comput. Sci.*, 285:187–243, Aug. 2002.

[10] D. Fallside and P. Walmsley. *XML Schema Part 0: Primer*, second edition, October 2004. http://www.w3.org/TR/xmlschema-0/.

[11] D. Florescu, A. Grünhagen, and D. Kossmann. Xl: an xml programming language for web service specification and composition. *Comput. Networks*, 42(5):641–660, 2003.

[12] M. Harren, M. Raghavachari, O. Shmueli, M. Burke, R. Bordawekar, I. Pechtchanski, and V. Sarkar. XJ: Facilitating XML processing in Java. In *WWW'05*, pages 278–287. ACM Press, May 2005.

[13] H. Hosoya and B. C. Pierce. XDuce: A statically typed XML processing language. *ACM Trans. Inter. Tech.*, 3(2):117–148, 2003.

[14] H. Hosoya, J. Vouillon, and B. C. Pierce. Regular expression types for XML. *ACM SIGPLAN Notices*, 35(9):11–22, 2000.

[15] E. B. Johnsen and O. Owe. An asynchronous communication model for distributed concurrent objects. *Software and Systems Modeling*, 6(1):35–58, Mar. 2007.

[16] E. B. Johnsen, O. Owe, and M. Arnestad. Combining active and reactive behavior in concurrent objects. In *Proc. of the Norwegian Informatics Conference (NIK'03)*, pages 193–204. Tapir, Nov. 2003.

[17] E. B. Johnsen, O. Owe, and E. W. Axelsen. A run-time environment for concurrent objects with asynchronous method calls. In N. Martí-Oliet, editor, *Proc. 5th International Workshop on Rewriting Logic and its Applications (WRLA'04), Mar. 2004*, volume 117 of *Electronic Notes in Theoretical Computer Science*, pages 375–392. Elsevier Science Publishers, Jan. 2005.

[18] E. B. Johnsen, O. Owe, and I. C. Yu. Creol: A type-safe object-oriented model for distributed concurrent systems. *Theoretical Computer Science*, 365(1–2):23–66, Nov. 2006.

[19] M. Kempa and V. Linnemann. On XML Objects. In *Informal Proceedings of the Workshop on Programming Language Technologies for XML (PLAN-X 2002), PLI 2002, Pittsburgh, USA*, pages 44–54, 3.-8. October 2002.

[20] C. Kirkegaard, A. Møller, and M. I. Schwartzbach. Static analysis of XML transformations in Java. *IEEE Transactions on Software Engineering*, 30(3):181–192, March 2004.

[21] E. Meijer, W. Schulte, and G. Bierman. Programming with circles, triangles and rectangles. In *Proceedings of the XML Conference*, 2003.

[22] A. Møller and M. Schwartzbach. The design space of type checkers for XML transformation languages. In *ICDT'05*, volume 3363 of *LNCS*, pages 17–36. Springer-Verlag, January 2005.

[23] M. Murata, D. Lee, and M. Mani. Taxonomy of XML schema languages using formal language theory. In *Extreme Markup Languages*, Montreal, Canada, 2001.

[24] D. Obasanjo. Overview of C_ω. `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnexxml/html/xml01142005.asp`, January 2005.

[25] D. A. Thomas. The impedance imperative - tuples + objects + infosets = too much stuff! *Journal of Object Technology*, 2(5):7–12, 2003.

[26] W3C (World Wide Web Consortium). *XML Path Language (XPath) Version 1.0*, 1999. W3C Recommendation 16 November 1999. `http://www.w3.org/TR/1999/REC-xpath-19991116`.

[27] W3C (World Wide Web Consortium). *XML Query Use Cases*, 2007. W3C Working Group Note 23 March 2007. `http://www.w3.org/TR/2007/NOTE-xquery-use-cases-20070323/`.

APPENDIX

A Execution of the Creol example

We here give an example of executing the program in Sec 5 with the following values:

The query DTD

```
<!ELEMENT query (title | author | editor | publisher)>
<!ELEMENT title  (#PCDATA )>
<!ELEMENT author  (#PCDATA )>
<!ELEMENT editor  (#PCDATA )>
<!ELEMENT publisher  (#PCDATA )>
```

in Creol syntax:

```
queryType =
xmlSchema("query",
  elemDecl("query" , elemCt("title" | ("author" | ("editor" | "publisher"))))
  elemDecl("title", elemCt(PCDATA))
  elemDecl("author", elemCt(PCDATA))
  elemDecl("editor", elemCt(PCDATA))
  elemDecl("publisher", elemCt(PCDATA)), noAttDecl) .
```

The query result DTD

This value is never used in the program but all query results conform to this DTD.

```
<!ELEMENT result (book | err_result)>
<!ELEMENT err_result  (#PCDATA )>
<!ELEMENT book  (title,  (author+ | editor+ ), publisher, price )>
<!ELEMENT title  (#PCDATA )>
<!ELEMENT author  (#PCDATA )>
<!ELEMENT editor  (#PCDATA )>
<!ELEMENT publisher  (#PCDATA )>
<!ELEMENT price  (#PCDATA )>
```

in Creol syntax:

```
resultType =
xmlSchema("result",
  elemDecl("result" , elemCt("book" | "err_result"))
  elemDecl("err_result", elemCt(PCDATA))
  elemDecl("book", elemCt("title" @ ("author"+ |"editor"+) @ "publisher" @ "price"))
  elemDecl("author", elemCt(PCDATA))
  elemDecl("editor", elemCt(PCDATA))
  elemDecl("title", elemCt(PCDATA))
  elemDecl("publisher", elemCt(PCDATA))
  elemDecl("price", elemCt(PCDATA)) .
```

The library catalogue
The catalogue of books is a modified version of the bibliography document used as an example in [27]:

```
<!DOCTYPE bib [
  <!ELEMENT bib  (book* )>
  <!ELEMENT book  (title,  (author+ | editor+ ), publisher, price )>
  <!ATTLIST book  >
  <!ELEMENT author (#PCDATA ) >
  <!ELEMENT editor  (#PCDATA )>
  <!ELEMENT title  (#PCDATA )>
  <!ELEMENT publisher  (#PCDATA )>
  <!ELEMENT price  (#PCDATA )>
  ]>
<bib>
  <book>
    <title>TCP/IP Illustrated</title>
    <author>Stevens, W.</author>
    <publisher>Addison-Wesley</publisher>
    <price>65.95</price>
  </book>
  <book>
    <title>Advanced Programming in the Unix environment</title>
    <author>Stevens, W.</author>
    <publisher>Addison-Wesley</publisher>
    <price>65.95</price>
  </book>
  <book>
    <title>Data on the Web</title>
    <author>Abiteboul, Serge</author>
    <author>Buneman, Peter</author>
    <author>Suciu, Dan</author>
    <publisher>Morgan Kaufmann Publishers</publisher>
    <price>39.95</price>
  </book>
  <book>
    <title>The Economics of Technology and Content for Digital TV</title>
    <editor>Gerbarg, Darcy</editor>
    <publisher>Kluwer Academic Publishers</publisher>
    <price>129.95</price>
  </book>
</bib>
```

The library catalogue as a Creol value is:

```
eq catalogue =
xmlDoc("bib"[
  ("book"[
    ("title"[tx("TCP/IP Illustrated")])
    ("author"[tx("Stevens, W.")])
    ("publisher"[tx("Addison-Wesley")])
    ("price"[tx("65.95")])])
```

```
  ("book"[
    ("title"[tx("Advanced Programming in the Unix environment")])
    ("editor"[tx("Stevens, W.")])
    ("publisher"[tx("Addison-Wesley")])
    ("price"[tx("65.95")])])
  ("book"[
    ("title"[tx("Data on the Web")])
    ("author"[tx("Abiteboul, Serge")])
    ("author"[tx("Buneman, Peter")])
    ("author"[tx("Suciu, Dan")])
    ("publisher"[tx("Morgan Kaufmann Publishers")])
    ("price"[tx("39.59")])])
  ("book"[
    ("title"[tx("The Economics of Technology and Content for Digital TV")])
    ("author"[tx("Gerbarg, Darcy")])
    ("publisher"[tx("Kluwer Academic Publishers")])
    ("price"[tx("129.95")])])],
xmlSchema("bib",
  elemDecl("bib", elemCt("book" * ))
  elemDecl("book", elemCt("title" @ ("author"+ |"editor"+) @ "publisher" @ "price"))
  elemDecl("author", elemCt(PCDATA))
  elemDecl("editor", elemCt(PCDATA))
  elemDecl("title", elemCt(PCDATA))
  elemDecl("publisher", elemCt(PCDATA))
  elemDecl("price", elemCt(PCDATA)),noAttDecl)) .
```

The client sends four queries to the server. In XML notation the queries and responses would look like this:

Query 1

```
<query><title>TCP/IP Illustrated</title><query>

<result>
  <book>
    <title>TCP/IP Illustrated</title>
      <author>Stevens, W.</author>
      <publisher>Addison-Wesley</publisher>
      <price>65.95</price>
  </book>
</result>
```

Query 2

```
<query><price>65.95</price><query>

<result>
  <err_result>Invalid query type</err_result>
</result>
```

Query 3

```
<query><publisher>Addison-Wesley</publisher><query>

<result>
  <book>
```

Chapter 11

Paper #4: Executable interface specifications for testing asynchronous Creol components

Immo Grabe, Martin Steffen, and Arild B. Torjusen

Published as Research Report 375, Dept. of Informatics, Univ. of Oslo, July 2008 (revised May 2010) [GST08].

UNIVERSITY OF OSLO
Department of Informatics

Executable interface specifications for testing asynchronous Creol components

Research Report 375

Immo Grabe

Martin Steffen

Arild B. Torjusen

ISBN 82-7368-335-4
ISSN 0806-3036

July 14, 2008
Revised May 2010

Executable Interface Specifications for Testing Asynchronous Creol Components*

Immo Grabe[1], Martin Steffen[2], and Arild B. Torjusen[2]

[1] Christian-Albrechts University Kiel, Germany
[2] University of Oslo, Norway

Abstract. We propose and explore a formal approach for black-box testing asynchronously communicating components in open environments. Asynchronicity poses a challenge for validating and testing components. We use Creol, a high-level, object-oriented language for distributed systems and present an interface specification language to specify components in terms of traces of observable behavior.
The language enables a concise description of a component's behavior, it is executable in rewriting logic and we use it to give test specifications for Creol components. In a specification, a clean separation between interaction under control of the component or coming from the environment is central, which leads to an assumption-commitment style description of a component's behavior. The assumptions schedule the inputs, whereas the outputs as commitments are tested for conformance with the specification. The asynchronous nature of communication in Creol is respected by testing only up-to a notion of observability. The existing Creol interpreter is combined with our implementation of the specification language to obtain a specification-driven interpreter for testing.

1 Introduction

To reason about open distributed systems and predicting their behavior is intrinsically difficult. A reason for that is the inherent asynchronicity and the resulting non-determinism. It is generally accepted that the only way to approach complex systems is to "divide-and-conquer", i.e., consider components interacting with their environment. Abstracting from internal executions, their black-box behavior is given by interactions at their *interface*. In this paper we use Creol [20], a programming and modeling language for distributed systems based on concurrent, active objects communicating via asynchronous method calls.

To describe and test Creol components, we introduce a concise specification language over communication labels. The expected behavior is given as a set of traces at the interface. Both input and output interactions are specified but play quite different roles. As input events are not under the control of the object, but

* Part of this work has been supported by the EU-project IST-33826 *Credo: Modeling and analysis of evolutionary structures for distributed services* and the German-Norwegian DAAD-NWO exchange project *Avabi* (Automated validation for behavioral interfaces of asynchronous active objects).

of the environment, input is considered as assumptions about the environment whereas output describes commitments of the object. For input interactions, we ensure that the specified assumptions on the environment are fulfilled by *scheduling* the incoming calls in the order specified, while for output events, which are controlled by the component, we *test* that the events occur as specified. An expression in the specification language thus gives an assumption-commitment style specification[11] for a component by defining the valid observable output behavior under the assumption of a certain scheduling of the input. Scheduling and testing of a component is done by synchronizing the execution of the component with the specification. As a result, the scheduling is enforced in the execution of the component and the actual outgoing interactions from the component are tested against the output labels in the specification. This gives a framework for testing whether an implementation of a component conforms with the interface specification. Incorrect or nonconforming behavior of the component under a given scheduling is reported as an error.

It is important in the specification, to carefully distinguish between the interactions which are scheduled and those for which the component is responsible and which are checked for conformance. We do so by formalizing *well-formedness* conditions on specifications. Well-formedness enforces a syntactic distinction between input and output specifications and, in addition, assures that only "meaningful" traces, i.e., those corresponding to possible behavior, can be specified. Besides that, the specification language captures two crucial features of the interface behavior of Creol objects. First, Creol allows to dynamically create objects and threads (via asynchronous method calls), which gives rise to dynamic scoping. This is reflected in the interface behavior by scope extrusion and the specification language allows to express *freshness* of communicated object and thread references. Second, due to the asynchronous nature of the communication model, the order in which outgoing messages from a component are observed by an external observer does not necessarily reflect the order in which they where actually sent. We take this asynchronous message passing into account by only considering trace specifications up-to an appropriate notion of *observational equivalence*.

Contributions The paper contains the following contributions: We *formalize* the interface behavior of a concurrent, object-oriented, language plus a corresponding behavioral interface specification language in Sect. 2 and Sect. 3. This gives the basis for testing active Creol objects, where a test environment can be simulated by execution of the specifications. Sect. 4 explains how to compose a Creol program and a specification and how to use this for testing. Furthermore, the existing Creol interpreter is extended with the *implementation* of the specification language. This yields a specification-driven interpreter for testing asynchronous Creol components. The implementation is described in Sect. 5

2 The Creol Language

Creol [10,20] is a high-level object-oriented language for distributed systems, featuring active objects and asynchronous method calls. Concentrating on the core features, we elide inheritance, dynamic class upgrades, etc. They would complicate the interface description, but not alter the basic ideas presented here.

The Creol-language features active objects and its communication model is based on exchanging messages *asynchronously.* This is in contrast with object-oriented languages based on multi-threading, such as *Java* or $C^{\#}$, which use "synchronous" message passing in which the calling thread inside one object blocks and control is transferred to the callee. Exchanging messages asynchronously decouples caller and callee, which makes that mode of communication advantageous in a distributed setting. On the receiver side, i.e., at the side of the callee, each object possesses an input "queue" in which incoming messages are waiting to be served by the object. To avoid uncontrolled interference, each object acts as a *monitor*, i.e., at most one method body is executing at each point in time. The choice, which method call in the input queue is allowed to enter the object next is *non-deterministic* (i.e., the term input "queue" is a slight misnomer, as it seems to indicate FiFo-discipline in the scheduling).

We start with the abstract syntax in Sect. 2.1. Afterwards, Sect. 2.2 contains the static typing rules and Sect. 2.3 the operational semantics.

2.1 Syntax

The abstract syntax of the calculus, which is in the style of standard object calculi [1], is given in Tab. 1. It distinguishes between *user* syntax and *run-time* syntax, the latter underlined. The user syntax contains the phrases in which programs are written; the run-time syntax contains syntactic material additionally needed to express the behavior of the executing program in the operational semantics. The latter are not found in a program written by the user, but generated at run-time by the rules of the operational semantics.

The basic syntactic category of names n, which count among the values v, represents references to classes, to objects, and to threads. To facilitate reading, we allow ourselves to write o and its syntactic variants for names referring to objects, c for classes, and n when being unspecific. Technically, the disambiguation between the different roles of the names is done by the type system and the abstract syntax of Tab. 1 uses the non-specific n for names. The unit value is represented by () and x stands for variables, i.e., local variables and formal parameters, but not instance variables.

A *component* C is a collection of classes, objects, and (named) threads, with $\mathbf{0}$ representing the empty component. The sub-entities of a component are composed using the parallel-construct $\|$. The entities executing in parallel are the named threads $n\langle t\rangle$, where t is the code being executed and n the name of the thread. The name n of the thread is, at the same time, the future reference under which the result value of t, if any[3], will be available. In this paper, when

[3] There will be no result value in case of non-terminating methods.

$C ::= \mathbf{0} \mid C \parallel C \mid \underline{\nu(n{:}T).C} \mid n[\![O]\!] \mid \underline{n[n, F, L]} \mid \underline{n\langle t\rangle}$		component
$O ::= F, M$		object
$M ::= l = m, \ldots, l = m$		method suite
$F ::= l = f, \ldots, l = f$		fields
$m ::= \varsigma(n{:}T).\lambda(x{:}T, \ldots, x{:}T).t$		method
$f ::= \varsigma(n{:}T).\lambda().v \mid \varsigma(n{:}T).\lambda().\bot_{n'}$		field
$t ::= v \mid \mathsf{stop} \mid \mathsf{let}\, x{:}T = e\, \mathsf{in}\, t$		thread
$e ::= t \mid \mathsf{if}\, v = v\, \mathsf{then}\, e\, \mathsf{else}\, e \mid \mathsf{if}\, \mathit{undef}(v.l())\, \mathsf{then}\, e\, \mathsf{else}\, e$		expr.
$\mid \; v@l(\vec{v}) \mid \underline{v.l(\vec{v})} \mid v.l() \mid v.l := \varsigma(s{:}n).\lambda().v$		
$\mid \; \mathsf{new}\, n \mid \underline{\mathsf{claim@}(n, n)} \mid \underline{\mathsf{get@}n} \mid \underline{\mathsf{suspend}(n)} \mid \underline{\mathsf{grab}(n)} \mid \underline{\mathsf{release}(n)}$		
$v ::= x \mid n \mid ()$		values
$L ::= \bot \mid \top$		lock status

Table 1. Abstract syntax

describing the interface behavior, we restrict ourselves to the situation where the component consists of one object only, plus arbitrary many threads/method bodies under execution. A class $c[\![O]\!]$ carries a name c and defines its methods and fields in O. An object $o[c, F, L]$ with identity o keeps a reference to the class c it instantiates, stores the current value F of its fields, and maintains a *binary lock* L indicating whether any code is currently active inside the object (in which case the lock is taken) or not (in which case the lock is free). The symbols $\top$ and $\bot$ indicate that the lock is taken or free respectively. Of the three kinds of entities at the component level—threads $n\langle t\rangle$, classes $n[\![O]\!]$, and objects $o[c, F, L]$—only the threads are *active*, executing entities, being the target of the reduction rules. The objects, in contrast, store the state in their fields or instance variables, whereas the classes are constant entities specifying the methods.

The named threads $n\langle t\rangle$ are incarnations of method bodies "in execution". Incarnations insofar as the formal parameters have been replaced by actual ones, especially the method's self-parameter has been replaced by the identity of the target object of the method call. The term t is basically a sequence of expressions, where the let-construct is used for sequencing and for local declarations.[4] During execution, $n\langle t\rangle$ contains in t the running code of a method body. When evaluated, the thread is of the form $n\langle v\rangle$ and the value can be accessed via n, the future reference, or future for short.

Each thread belongs to one specific object "inside" which it executes, i.e., whose instance variables it has access to. Object locks are used to rule out unprotected concurrent access to the object states: Though each object may have more than one method body incarnation partially evaluated, at each time point at most one of those bodies (the lock owner) can be active inside the object. In the terminology of Java, all methods are implicitly considered "synchronized". The crucial difference between Java's multi-threading concurrency model and

[4] $t_1; t_2$ (sequential composition) abbreviates $\mathsf{let}\, x{:}T = t_1\, \mathsf{in}\, t_2$, where x does not occur free in t_2.

Creol's active objects model used here is the way method calls are issued at the caller site. In Java and similar languages, method calls are *synchronous* in the sense that the calling activity blocks to wait for the return of the result and thus the control is transferred to the callee. Method calls in Creol are issued *asynchronously,* i.e., the calling thread continues executing and the code of the method being called is computed concurrently in a new thread located in the callee object. In that way, a method call never transfers control from one object, the caller, to another one, the callee. In other words, no thread ever crosses the boundaries of an object, which means, the boundaries of an object are at the same time boundaries of the threads and thus, the objects are at the same time units of concurrency. Thus, the objects are harnessing the activities and can be considered as bearers of the activities. This is typical for object-oriented languages based on *active objects*. The ν-operator is used for hiding and dynamic scoping, as known from the π-calculus [23]. In a component $C = \nu(n{:}T).C'$, the scope of the name n (of type T) is restricted to C' and unknown outside C. ν-binders are introduced when dynamically creating new named entities, i.e., when instantiating new objects or new threads. The scope of a ν-binder is dynamic, when the name is communicated by message passing, the scope is enlarged.

Besides components, the grammar specifies the lower level syntactic constructs, in particular, methods, expressions, and (unnamed) threads, which are basically sequences of expressions. A method $\varsigma(s{:}T).\lambda(\vec{x}{:}\vec{T}).t$ provides the method body t abstracted over the ς-bound "self" parameter, here s, and the formal parameters $\vec{x}$. For uniformity, fields are represented as methods without parameters (except self), with a body being either a value or yet undefined. Note that the methods are stored in the classes but the fields are kept in the objects. In freshly created objects, the lock is free, and all fields carry the undefined reference $\perp_c$, where class name c is the (return) type of the field.

We use f for instance variables or fields and $l = \varsigma(s{:}T).\lambda().v$, resp. $l = \varsigma(s{:}T).\lambda().\perp_c$ for field variable definition (l is the label of the field). Field access is written as $v.l()$ and field update as $v'.l := \varsigma(s{:}T).\lambda().v$. By convention, we abbreviate the latter constructs by $l = v$, $l = \perp_c$, $v.l$, and $v'.l := v$. Note that the construct $v.l()$ is used for field access only, but not for method invocation. The expression $v@l(\vec{v})$ denotes an asynchronous method call, $v.l(\vec{v})$ is run-time syntax for a synchronous call and hence not available for the user. We also use $v_\perp$ to denote either a value v or a symbol $\perp_c$ for being undefined. Note that the syntax does not allow to set a field back to undefined. Direct access (read or write) to fields across object boundaries is forbidden by convention, and we do not allow method update. Instantiation of a new object from class c is denoted by new c.

The expression $o@l(\vec{v})$ denotes an asynchronous method call, where the caller creates a new thread/future reference and continues its execution. The further expressions claim, get, suspend, grab, and release deal with synchronization. As mentioned, objects come equipped with binary locks, responsible for assuring mutual exclusion. The two basic, complementary operations on a lock are grab and release. The first allows an activity to acquire access in case the lock is free

($\bot$), thereby setting it to $\top$, and release(o) conversely relinquishes the lock of the object o, giving other threads the chance to be executed in its stead, when succeeding to grab the lock via grab(o). The user is not allowed to directly manipulate the object locks. Thus, both expressions belong to the run-time syntax, underlined in Tab. 1, and are only generated and handled by the operational semantics as auxiliary expression at run-time. Instead of using directly grab and release, the lock-handling is done automatically when executing a method body: before starting to execute, the lock has to be acquired and upon termination, the lock is released again. Besides that, lock-handling is involved also when results are claimed, i.e., when a client code executing in an object, say o, intends to read the result of a future. The expression claim@(n, o) is the attempt to obtain the result of a method call from the future n while in possession of the lock of object o. There are two possibilities in that situation: either the value of the future has already been determined, i.e., the method calculating the result has terminated, in which case the client just obtains the value *without* loosing its own lock. In the alternative case, where the value is not yet determined, the client trying to read the value gives up its lock via release and continues executing only after the requested value has been determined (using get to read it) and after it has re-acquired the lock. Unlike claim, the get-operation is not part of the user-syntax. Both expressions are used to read back the value from a future and the difference in behavior is that get unconditionally attempts to get the value, i.e., blocks until the value has arrived, whereas claim gives up the lock temporarily, if the value has not yet arrived, as explained. Note the order in which get and grab are executed after releasing the lock: the value is read in via get *before* the lock has actually been re-acquired! We assume by convention, that when appearing in methods of classes, the claim- and the suspend-command only refer to the self-parameter *self*, i.e., they are written claim@(n, self) and suspend(self).

2.2 Typing

The Creol calculus presented above is strongly typed. The static type system is rather conventional and coincides largely with the one given in [27]. We included the rules, for the sake of completeness, in the appendix of this technical report, in Tabs. 13 and 14, without much explanations. The typing judgments are of the following form: $\Delta \vdash C : \Theta$ on the level of components (cf. Tab. 13) asserts well-typedness of C under the assumption name context Δ and with commitments Θ. On the level of threads, expressions, and their sub-phrases, $\Gamma; \Delta \vdash t : T$ asserts well-typedness of thread t with type T, under the name assumptions Δ and variable assumptions Γ.

2.3 Operational Semantics

The operational semantics of a program being tested is given in two stages: steps *internal* to the program, and those occurring at the interface. The two stages correspond to the rules of Tab. 2 and 5. The internal rules of Tab. 2 deal with steps not interacting with the object's environment, such as sequential

$n\langle \mathsf{let}\ x{:}T = v\ \mathsf{in}\ t\rangle \rightsquigarrow n\langle t[v/x]\rangle$ RED

$n\langle \mathsf{let}\ x_2{:}T_2 = (\mathsf{let}\ x_1{:}T_1 = e_1\ \mathsf{in}\ e)\ \mathsf{in}\ t\rangle \rightsquigarrow n\langle \mathsf{let}\ x_1{:}T_1 = e_1\ \mathsf{in}\ (\mathsf{let}\ x_2{:}T_2 = e\ \mathsf{in}\ t)\rangle$ LET

$n\langle \mathsf{let}\ x{:}T = (\mathsf{if}\ v = v\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2)\ \mathsf{in}\ t\rangle \rightsquigarrow n\langle \mathsf{let}\ x{:}T = e_1\ \mathsf{in}\ t\rangle$ COND_1

$n\langle \mathsf{let}\ x{:}T = (\mathsf{if}\ v_1 = v_2\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2)\ \mathsf{in}\ t\rangle \rightsquigarrow n\langle \mathsf{let}\ x{:}T = e_2\ \mathsf{in}\ t\rangle$ COND_2

$n\langle \mathsf{let}\ x{:}T = (\mathsf{if}\ \mathsf{undef}(\bot_{c'})\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2)\ \mathsf{in}\ t\rangle \rightsquigarrow n\langle \mathsf{let}\ x{:}T = e_1\ \mathsf{in}\ t\rangle$ $\text{COND}_1^{\bot}$

$n\langle \mathsf{let}\ x{:}T = (\mathsf{if}\ \mathsf{undef}(v)\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2)\ \mathsf{in}\ t\rangle \rightsquigarrow n\langle \mathsf{let}\ x{:}T = e_2\ \mathsf{in}\ t\rangle$ $\text{COND}_2^{\bot}$

$n\langle \mathsf{let}\ x{:}T = \mathsf{stop}\ \mathsf{in}\ t\rangle \rightsquigarrow n\langle \mathsf{stop}\rangle$ STOP

$o[c, F, L] \parallel n\langle \mathsf{let}\ x{:}T = o.l()\ \mathsf{in}\ t\rangle \xrightarrow{\tau} o[c, F, L] \parallel n\langle \mathsf{let}\ x{:}T = F.l(o)()\ \mathsf{in}\ t\rangle$ FLOOKUP

$o[c, F, L] \parallel n\langle \mathsf{let}\ x{:}T = o.l := v\ \mathsf{in}\ t\rangle \xrightarrow{\tau} o[c, F.l := v, L] \parallel n\langle \mathsf{let}\ x{:}T = o\ \mathsf{in}\ t\rangle$ FUPDATE

$n\langle \mathsf{let}\ x : T = o@l(\vec{v})\ \mathsf{in}\ t\rangle \rightsquigarrow$
$\nu(n'{:}T)(n\langle \mathsf{let}\ x : T = n'\ \mathsf{in}\ t\rangle \parallel n'\langle \mathsf{let}\ x : T = o.l(\vec{v})\ \mathsf{in}\ \mathsf{stop}\rangle)$ CALLO_i

$n_1\langle v\rangle \parallel n_2\langle \mathsf{let}\ x : T = \mathsf{claim}@(n_1, o)\ \mathsf{in}\ t\rangle \rightsquigarrow n_1\langle v\rangle \parallel n_2\langle \mathsf{let}\ x : T = v\ \mathsf{in}\ t\rangle$ CLAIM_i^1

$$\frac{t_2 \neq v}{n_2\langle t_2\rangle \parallel n_1\langle \mathsf{let}\ x : T = \mathsf{claim}@(n_2, o)\ \mathsf{in}\ t_1'\rangle \rightsquigarrow n_2\langle t_2\rangle \parallel n_1\langle \mathsf{let}\ x : T = \mathsf{release}(o); \mathsf{get}@n_2\ \mathsf{in}\ \mathsf{grab}(o); t_1'\rangle}\ \text{CLAIM}_i^2$$

$n_1\langle v\rangle \parallel n_2\langle \mathsf{let}\ x : T = \mathsf{get}@n_1\ \mathsf{in}\ t\rangle \rightsquigarrow n_1\langle v\rangle \parallel n_2\langle \mathsf{let}\ x : T = v\ \mathsf{in}\ t\rangle$ GET_i

$n\langle \mathsf{suspend}(o); t\rangle \rightsquigarrow n\langle \mathsf{release}(o); \mathsf{grab}(o); t\rangle$ SUSPEND

$o[c, F, \bot] \parallel n\langle \mathsf{grab}(o); t\rangle \xrightarrow{\tau} o[c, F, \top] \parallel n\langle t\rangle$ GRAB

$o[c, F, \top] \parallel n\langle \mathsf{release}(o); t\rangle \xrightarrow{\tau} o[c, F, \bot] \parallel n\langle t\rangle$ RELEASE

Table 2. Internal steps

composition, conditionals, field lookup and update, etc. The rules are standard and fairly straightforward, and we show them for reference only. The steps are given as unlabelled steps, where we distinguish between $\rightsquigarrow$-steps (confluent) and $\xrightarrow{\tau}$-steps (non-confluent, accessing the instance state) If the distinction does not play a role, we write $\rightarrow$. Components and the reduction relation are interpreted up-to standard structural congruences (cf. Tab. 15 and 16 in the appendix). We write $\Longrightarrow$ for the reflexive and transitive closure of the internal steps from Tab. 16. The communication labels, the basic building blocks of the interface interactions, are given in Tab. 3. A component or object exchanges information with the environment via *call*- and *return*-labels, and the interactions is either incoming or outgoing (marked ? resp. !). The basic label $n\langle \mathit{call}\ o.l(\vec{v})\rangle$ represents a call of method l in object o. In that label, n is a name identifying the thread that executes the method in the callee and is therefore the (future) reference under which the result of the method call will be available (if ever) for the caller. The

incoming label $n\langle return(v)\rangle?$ hands the value from the corresponding call back to the object, which renders it ready to be read. Its counterpart, the outgoing return, passes the value to the environment. Besides that, labels can be prefixed by bindings of the form $\nu(n{:}T)$ which express freshness of the transmitted name, i.e., scope extrusion. As usual, the order of such bindings does not play a role

Given a basic label $\gamma = \nu(\Xi).\gamma'$ where Ξ is a name context such that $\nu(\Xi)$ abbreviates a sequence of single $n{:}T$ bindings (whose names are assumed all disjoint, as usual) and where γ' does not contain any binders, we call γ' the *core* of the label and refer to it by $\lfloor\gamma\rfloor$. We define the core analogously for receive and send labels. The free names $fn(a)$ and the bound names $bn(a)$ of a label a are defined as usual, whereas $names(a)$ refer to all names of a.

The interface behavior is given by the 4 rules of Tab. 5, which correspond to the 4 different kinds of labels, a call or a return, either incoming or outgoing. The external steps are given as transitions of the form $\Xi \vdash C \xrightarrow{a} \acute{\Xi} \vdash \acute{C}$, where Ξ and $\acute{\Xi}$ represents the assumption/commitment contexts of C before and after the step, respectively. In particular, the context contains the identities of the objects and threads known so far, and the corresponding typing information. An important, but standard, part of the external semantics is to check the static *typing* assumptions, e.g., whether at most the names actually occurring in the core of the label are mentioned in the ν-binders of the label and whether the transmitted values are of the correct types. Besides *checking* whether the assumptions are met before a transition, the contexts are *updated* by a transition step. These two operations are captured by the following notation

$$\Xi \vdash a : T \qquad \text{and} \qquad \Xi + a \tag{1}$$

which constitute part of the rules' premises in Tab. 5. Intuitively, they mean the following: label a is well-formed and well-typed wrt. the information Ξ and refers to an asynchronous call which results in a value of type T. If not interested in the type, we write $\Xi \vdash a : ok$, instead. The right-hand notation of (1) extends the binding context Ξ by the bindings transmitted as part of label a appropriately. The formal definition of context update is given below and checking of static typing assumptions is defined as follows:

Definition 1 (Well-formedness and well-typedness). *A label $a = \nu(\Xi).\lfloor a\rfloor$ is* well-formed, *written $\vdash a$, if $dom(\Xi) \subseteq names(\lfloor a\rfloor)$ and if Ξ is a well-formed name-context for object and future names, i.e., no name bound in Ξ occurs twice. The assertion*

$$\acute{\Xi} \vdash o.l? : \vec{T} \to T \tag{2}$$

$\gamma ::= n\langle call\ n.l(\vec{v})\rangle \mid n\langle \mathsf{return}(n)\rangle \mid \nu(n{:}T).\gamma$	basic labels
$a ::= \gamma? \mid \gamma!$	input and output labels

Table 3. Communication labels

$$\frac{\acute{\Xi} \vdash n : [T] \quad ; \acute{\Xi} \vdash \vec{v} : \vec{T} \quad a = n\langle call\ o.l(\vec{v})\rangle?}{\acute{\Xi} \vdash a : \ \vec{T} \to \_} \text{LT-CallI}$$

$$\frac{; \acute{\Xi} \vdash v : T \quad a = n\langle return(v)\rangle?}{\acute{\Xi} \vdash a : \ \_ \to T} \text{LT-RetI}$$

Table 4. Typechecking labels

("an incoming call of the method labeled l in object o expects arguments of type $\vec{T}$ and results in a value of type T") is given by the following rule, i.e., implication:

$$\frac{; \acute{\Theta} \vdash o : c \qquad ; \acute{\Xi} \vdash c : [\![\ldots, l{:}\vec{T} \to T, \ldots]\!]}{\acute{\Xi} \vdash o.l? : \vec{T} \to T} \tag{3}$$

For outgoing calls, $\acute{\Xi} \vdash o.l! : \vec{T} \to T$ is defined dually. In particular, in the first premise, $\acute{\Theta}$ is replaced by $\acute{\Delta}$. Well-typedness *of an incoming core label a with expected type $\vec{T}$, resp., T, and relative to the name context $\acute{\Xi}$ is asserted by*

$$\acute{\Xi} \vdash a : \vec{T} \to \_ \quad \textit{resp.,} \quad \acute{\Xi} \vdash a : \_ \to T \ , \tag{4}$$

as given by Tab. 4.

Note that the receiver o of the call is checked using only the commitment context $\acute{\Theta}$, to assure that o is a component object. Note further that to check the interface type of the class c, the full $\acute{\Xi}$ is consulted, since the argument types $\vec{T}$ or the result type T may refer to both component and environment classes.

As mentioned, the contexts are *updated* by a transition step. especially they are extended by the new names whose scope is extruded. For the binding part Ξ' of a label $\nu(\Xi').\gamma$, the scope of the references to existing objects and thread names Δ' extrudes across the border. In the step, Δ' extends the assumption context Δ and Θ' the commitment context Θ. This gives rise to the following definition.

Definition 2 (Context update). *Let Ξ be a name context and $a = \nu(\Xi').\lfloor a \rfloor$ an* incoming *label. Then the definitions of the post-contexts $\acute{\Delta}$ and $\acute{\Theta}$ are given as follows, when $n{:}[T]$ is the binding for the thread name:*

$$\acute{\Delta} = (\Delta, \Xi') \setminus n{:}T \quad \textit{and} \quad \acute{\Theta} = \Theta, n{:}T \ . \tag{5}$$

We write $\Xi + a$ for that update. For outgoing *communication, the definition is applied dually.*

Given the definitions for well-typedness and context update, we describe the rules of Tab. 5. Rule CallI deals with incoming calls, and basically adds the new thread n (which at the same time represents the future reference for the

$$\frac{a = \nu(\Xi').\ n\langle \mathit{call}\ o.l(\vec{v})\rangle?\quad \Xi \vdash a : T\qquad \acute{\Xi} = \Xi + a}{\Xi \vdash C \parallel o[c, F, \bot] \xrightarrow{a} \acute{\Xi} \vdash C \parallel o[c, F, \top] \parallel n\langle \mathsf{let}\ x{:}T = M.l(o)(\vec{v})\ \mathsf{in}\ \mathsf{release}(o); x\rangle}\ \textsc{CallI}$$

$$\frac{a = \nu(\Xi').\ n\langle \mathit{call}\ o.l(\vec{v})\rangle!\quad \Xi' = \mathit{fn}(\lfloor a \rfloor) \cap \Xi_1\quad \acute{\Xi}_1 = \Xi_1 \setminus \Xi'\quad \Delta \vdash o\quad \acute{\Xi} = \Xi + a}{\Xi \vdash \nu(\Xi_1).(C \parallel n\langle \mathsf{let}\ x{:}T = o.l(\vec{v})\ \ \mathsf{in}\ \ t\rangle) \xrightarrow{a} \acute{\Xi} \vdash \nu(\acute{\Xi}_1).(C)}\ \textsc{CallO}$$

$$\frac{a = \nu(\Xi').\ n\langle \mathit{return}(v)\rangle?\qquad \Xi \vdash a : \mathit{ok}\qquad \acute{\Xi} = \Xi + a}{\Xi \vdash C \xrightarrow{a} \acute{\Xi} \vdash C \parallel n\langle v\rangle}\ \textsc{RetI}$$

$$\frac{a = \nu(\Xi').\ n\langle \mathit{return}(v)\rangle!\qquad \Xi' = \mathit{fn}(\lfloor a \rfloor) \cap \Xi_1\quad \acute{\Xi}_1 = \Xi_1 \setminus \Xi'\quad \acute{\Xi} = \Xi + a}{\Xi \vdash \nu(\Xi_1).(C \parallel n\langle v\rangle) \xrightarrow{a} \acute{\Xi} \vdash \nu(\acute{\Xi}_1).C}\ \textsc{RetO}$$

Table 5. External steps

eventual result) in parallel with the rest of the program. In the configuration after the reduction step, the meta-mathematical notation $M.l(o)(\vec{v})$ stands for $t[o/s][\vec{v}/\vec{x}]$, when the method suite $[M]$ equals $[\ldots, l = \varsigma(s{:}T).\lambda(\vec{x}{:}\vec{T}).t, \ldots]$. Note that the step is only possible, if the lock of the object is free ($\bot$); after the step, the lock is taken ($\top$). Rule CallO deals with outgoing calls. Remember that an asynchronous call, as given in $\textsc{CallO}_i$ from Tab. 2, does not immediately lead to an interface interaction, but is an internal step, which only afterwards (asynchronously) leads to the interface interaction as specified in CallO. Thus the t in the consequence of the rule always equals stop and the named thread n serves only to issue the outgoing call. Furthermore, the binding context Ξ is updated and, additionally, previously private names mentioned in Ξ_1 might escape by scope extrusion, which is calculated by the second and third premise. Rules RetI and RetO deal with returning the value at the end of a method call.

A trace of a well-typed component is a sequence of external steps; we write $\Xi_1 \vdash C_1 \overset{t}{\Longrightarrow} \Xi_2 \vdash C_2$ when the component $\Xi_1 \vdash C_1$ evolves to $\Xi_2 \vdash C_2$ by executing the trace t. The corresponding rules are given in Tab. 17. For $\Xi_1 \vdash C_1 \overset{\epsilon}{\Longrightarrow} \Xi_2 \vdash C_2$, we write shorter $\Xi_1 \vdash C_1 \Longrightarrow \Xi_2 \vdash C_2$, where ϵ denote the empty trace.

Remark 1. The rules for external steps from Tab. 5 resemble the ones given in [2]. There two differences are as follows. As we decided not to consider first-class futures and promises here (as in [2]), the set of rules is simpler here; rules dealing with obtaining the result of a future across the interface are not needed here. The second difference concerns the treatment of incoming calls in rule CallI. Here, an incoming call crossing the interface *atomically* grabs the lock, as we intend to describe and schedule the behavior and order the message communication in

the order they are executed in the object. Thus, the object's input queue is not modeled here. This is in contrast to the formalization in [2]. □

3 A Behavioral Interface Specification Language

The behavior of an object (or a component consisting of a set of objects, for that matter) at the interface is described by a sequence of labels as given by Tab. 3. The black-box behavior of a component can therefore be described by a set of *traces*, each consisting of a finite sequence of labels. To specify sets of label traces, we employ a simple trace language with prefix, choice and recursion. Table 6 contains its syntax. The syntax of the labels in the specification language,

$\gamma ::= x\langle \mathit{call}\ x.l(\vec{x})\rangle \mid x\langle \mathsf{return}(x)\rangle \mid \nu(x{:}T).\gamma \mid (x{:}T).\gamma$	basic labels
$a ::= \gamma? \mid \gamma!$	input and output labels
$\varphi ::= X \mid \epsilon \mid a.\varphi \mid \varphi + \varphi \mid \mathsf{rec}\ X.\varphi$	specifications

Table 6. Specification language

naturally, quite resembles the labels of Tab. 3. Comparing Tabs. 3 and 6, there are two differences: first, instead of names or references n, the specification language here uses variables. Second, the labels here allow a binding of the form $(x{:}T).\gamma$, which has no analog in Tab. 3; the form $\nu(x{:}T).\gamma$ corresponds to $\nu(n{:}T).\gamma$, of course. Both binding constructs act as variable declarations, with the difference that $\nu(x{:}T).\gamma$ not just introduces a variable (together with its type T), but in addition asserts that the names represented by that variable must be fresh. The binding $(x{:}T).\gamma$ corresponds to a conventional variable declaration, introducing the variable x which represents arbitrary values (of type T), either fresh or already known.

In the specification, it is important to distinguish between input and output interactions, as input messages are under the control of the environment, whereas the outputs are to be provided by the object as specified. This splits the specification into an *assumption* part under the responsibility of the environment, and a commitment part, controlled by the component. Hence, the input interactions are the ones being *scheduled,* whereas the outputs are not; they are used for *testing* that the object behaves correctly. To specify non-deterministic behavior, the language supports a choice operator, and we distinguish between choices taken by the environment—external choice—and those the object is responsible for—internal choice. Especially, we do not allow so-called mixed choice, i.e., choices are either under control of the object itself and concerns outgoing communication, or under control of the environment and concerns incoming communication. These restrictions are formalized next as part of the well-formedness conditions.

Example 1. We give an example to illustrate the scoping.

$$\nu(n_1{:}[T_1])\nu(o_2{:}c_2)n_1\langle \mathit{call}\ o_2.l()\rangle!\ .\ \nu(n_2{:}[T_2])(o_3{:}c_2)n_2\langle \mathit{call}\ o_1.l'(o_3)\rangle?$$

The specification begins with o_1 calling method l of o_2. As being the first step both objects o_1 and o_2 and the future reference n_1 are new. After the initial call we expect a call from o_2 to o_1. This call will be made by the new thread n_2. We expect o_3 to be the parameter of this call. Since o_3 is given as variable it might be either new or old. □

The programming language Creol is strongly typed. Accordingly, also the interface specification language sets value on the fact to allow only specifications that "make sense" type-wise. It makes, e.g., no sense to specify traces that insist on transmitting values in method calls that do not fit to the expected values as declared in the type of the corresponding method. Such specifications are rejected as being ill-typed and the restriction is justified by the fact that no component (which is assumed to be well-typed) can produce an ill-typed trace. This fact will later be proved. Indeed, well-typedness is an important part of the general well-formedness conditions we impose on the specifications. The typing conditions are rather standard and we mostly elide the rules for typing. They resemble closely the ones of [27] and especially from [2] for legality of traces. The difference to the first case is that [27] deals with a calculus for Java instead of Creol, and the latter [2] is more complex than the typing as considered here, as it deals additionally with the concept of first-class futures and promises. In general, the close resemblance wrt. typing is not surprising: the earlier papers dealt with the interface behavior in forms of sets of traces, which is here generalized to a more expressive recursive language to specify such behavior.

Remark 2. As mentioned, an important distinction in the trace specification is the one between incoming communication and outgoing. Especially, we do not allow *mixed choices,* i.e., a choice $\varphi_1 + \varphi_2$ where, e.g., φ_1 starts with an input and φ_2 with an output. This distinction could be enforced syntactically, for instance by distinguishing syntactically between external and internal choices. Such a syntactic distinction is often found for instance in formalizations based on *session types* [16,28], which can be seen a behavioral, trace-based interface description formulated by type system (therefore the term "session type" and not "session trace" . . .). Here we do not, however, reflect the distinction in the grammar of Tab. 6. Instead, the well-formedness conditions later discriminate between traces that start with an incoming communications and those starting with an output. □

3.1 Well-formedness

The grammar given in Tab. 6 allows to specify sets of traces. Not all specifications, however, are meaningful, i.e., describe traces actually possible at the interface of a component. We therefore formalize conditions to rule out such ill-formed specification where the main restrictions are: *Typing:* Values handed over must

$$\frac{}{\Xi \vdash \epsilon : wf^{?!}}\ \text{WF-EMTPY} \qquad \frac{\Xi \vdash X}{\Xi \vdash X : wf^{?!}}\ \text{WF-VAR}$$

$$\frac{a = \nu(\Xi').n\langle call\ o.l(\vec{v})\rangle? \qquad \Xi \vdash a : ok \qquad \acute{\Xi} = \Xi + a \qquad \acute{\Xi} \vdash \varphi : wf^{p}}{\Xi \vdash a.\varphi : wf^{?}}\ \text{WF-CALLI}$$

$$\frac{a = \nu(\Xi').n\langle return(v)\rangle? \qquad \Xi \vdash a : ok \qquad \acute{\Xi} = \Xi + a \qquad \acute{\Xi} \vdash \varphi : wf^{p}}{\Xi \vdash a.\varphi : wf^{?}}\ \text{WF-RETI}$$

$$\frac{\Xi \vdash \varphi_1 : wf^{p} \qquad \Xi \vdash \varphi_2 : wf^{p}}{\Xi \vdash \varphi_1 + \varphi_2 : wf^{p}}\ \text{WF-CHOICE} \qquad \frac{\Xi, X \vdash \varphi : wf^{p}}{\Xi \vdash \mathsf{rec}\ X.\varphi : wf^{p}}\ \text{WF-REC}$$

Table 7. Well-formedness of trace specifications

correspond to the expected types for that methods. *Scoping:* Variables must be declared (together with their types) before their use. *Communication patterns:* No value can be returned before a matching outgoing call has been seen at the interface. Specifications adhering to these restrictions are called *well-formed.*

Well-formedness is given straightforwardly by structural induction by the rules of Tab. 7. The rules formalize a judgment of the form

$$\Xi \vdash \varphi : wf^{p} \tag{6}$$

which stipulates φ's well-formedness under the assumption context Ξ. The meta-variable p (for polarity) stands for either ?, !, or ?!, where ?! indicates the polarity for an empty sequence or for a process variable, and ? and ! indicate well-formed input and output specifications respectively. As before, Ξ contains bindings from variables and class names to their types. The class names are considered as constants and also, the context Ξ will remain unchanged during the well-formedness derivation, since all classes are assumed to be known in advance and class names cannot be communicated. This is in contrast to the variables, which represent object references and references to future variables (resp. thread names). Besides that, the context also stores process variables X. The rules work as follows: The empty trace is well-formed (cf. rule WF-EMPTY), and a process variable X is well-formed, provided it had been declared before (written $\Xi \vdash X$, cf. rule WF-VAR). We omit the rules WF-CALLO and WF-RETO for outgoing calls, resp. outgoing get-labels, as they are dual to WF-CALLI and WF-RETI.

Remark 3 (Regular expressions). The specification language as given in Tab. 6 uses label-prefixing to express sequentiality in a trace and recursion to represent infinite behavior. Specifications thus resemble an automata-like or process-algebra representation. An alternative design is to specify sets of traces using the syntax of regular languages, i.e., to allow sequential composition $\varphi_1; \varphi_2$ of two formulas and to use iteration φ^* for infinite behavior.

Using regular expressions as sketched would slightly complicate the formulation of the well-formedness conditions. To accommodated for general sequential composition (in contrast to simple prefixing), the judgment for well-formedness would need to mention also the context *after* the traces given by the formula. I.e., the well-formedness judgment of equation (6) would have to be generalized to

$$\Xi \vdash \varphi : wf^p :: \acute{\Xi} \ , \tag{7}$$

where $\acute{\Xi}$ is the mentioned context after φ. Besides that, care must be taken wrt. *scope* of the variables. For instance, in $(\varphi_1 + \varphi_2); \varphi_3$, the trailing φ_3 may only use variables that have been introduced *both* in φ_1 and φ_2. In other words, the scope of a variable introduced in φ_1, say, does not extend unconditionally to φ_2. In a similar spirit and given $\varphi_1^+; \varphi_2$ scope of variables introduced in φ_1 *ends* at then end of φ_1 and does not extend to φ_2.

Both representations, the one of Tab. 6 and the one sketched here based on regular expressions, are equally expressive. For the sake of simplicity for the formalization, especially concerning the well-formedness conditions, we base this paper on the one using label-prefixing and explicit recursion. In the examples we sometimes use the regular expression syntax instead. □

3.2 Observational Blur

Creol objects communicate asynchronously and the order of messages might not be preserved during communication. Thus, an outside observer or tester can not see messages in the order in which they had been sent, and we need to relax the specification up-to some appropriate notion of *observational equivalence*, denoted by $\equiv_{obs}$ and defined by the rules of Tab. 8. Rule Eq-Switch captures the asynchronous nature of communication, in that the order of outgoing communication does not play a role. The definition corresponds to the one given in [27] and also of [18], in the context of multi-threading concurrency. Rule Eq-Plus allows to distribute an output over a non-deterministic choice, *provided* that it's a choice itself over outputs, as required by the well-formed condition in the premise. Rule Eq-Req finally expresses the standard unrolling of recursive definitions. Eq-Plus-Comm expresses commutativity of choice.

Next we state that well-formedness is preserved under the given equivalence.

Lemma 1. *If* $\Xi \vdash \varphi : wf^p$ *and* $\varphi \equiv_{obs} \varphi'$*, then* $\Xi \vdash \varphi' : wf^p$*.*

Proof. By induction on the rules of Tab. 7. Note that rule Eq-Plus explicitly requires output well-formedness of $\varphi_1 + \varphi_2$ it its premise. □

Given the equivalence relation, the meaning of a specification is given operationally by the rather obvious reduction rules of Tab. 9. The next lemmas express simple properties of the well-formedness condition, connecting it to the reduction relation.

Lemma 2. *Assume* $\Xi \vdash \varphi : wf$*.*

$$\frac{}{\nu(\Xi).\gamma_1!.\gamma_2!.\varphi \equiv_{obs} \nu(\Xi).\gamma_2!.\gamma_1!.\varphi}\ \text{Eq-Switch} \qquad \frac{\vdash (\varphi_1 + \varphi_2) : wf^!}{\gamma!.(\varphi_1 + \varphi_2) \equiv_{obs} \gamma!.\varphi_1 + \gamma!.\varphi_2}\ \text{Eq-Plus}$$

$$\mathsf{rec}\ X.\varphi \equiv_{obs} \varphi[\mathsf{rec}\ X.\varphi/X] \qquad \text{Eq-Rec}$$

$$\varphi_1 + \varphi_2 \equiv_{obs} \varphi_2 + \varphi_1 \qquad \text{Eq-Plus-Comm} \qquad\qquad \varphi + \epsilon \equiv_{obs} \varphi \qquad \text{Eq-Plus-Empty}$$

Table 8. Observational equivalence

$$\frac{\acute{\Xi} = \Xi + a}{\Xi \vdash a.\varphi \xrightarrow{a} \acute{\Xi} \vdash \varphi}\ \text{R-Pref} \qquad \frac{\Xi \vdash \varphi_1 \xrightarrow{a} \acute{\Xi} \vdash \varphi_1'}{\Xi \vdash \varphi_1 + \varphi_2 \xrightarrow{a} \acute{\Xi} \vdash \varphi_1'}\ \text{R-Plus}_1$$

$$\frac{\varphi \equiv_{obs} \varphi' \qquad \Xi \vdash \varphi' \xrightarrow{a} \Xi \vdash \varphi''}{\Xi \vdash \varphi \xrightarrow{a} \Xi \vdash \varphi''}\ \text{R-Equiv}$$

Table 9. φ rules

1. *Exactly one of the three conditions holds:* $\Xi \vdash \varphi : wf^{?!}$, $\Xi \vdash \varphi : wf^?$, *or* $\Xi \vdash \varphi : wf^!$
2. *If* $\varphi \xrightarrow{a}$ *with* a *an input, then* $\Xi \vdash \varphi : wf^?$. *Dually for outputs.*
3. *If* $\Xi \vdash \varphi : wf^?$, *then* $\varphi \xrightarrow{a}$ *with* a *an input. Dually for outputs.*

Proof. Part 1 by straightforward induction on the rules of Tab. 7. Part 2 by inverting the rules of 9 and by inspection of the rules for well-formedness. Part 3 works similarly.

Lemma 3 (Subject reduction). $\Xi \vdash \varphi : wf$ *and* $\Xi \vdash \varphi \xrightarrow{a} \acute{\Xi} \vdash \acute{\varphi}$, *then* $\acute{\Xi} \vdash \acute{\varphi} : wf$.

Proof. By straightforward induction on derivations of the rules of Tab. 9. □

Lemma 4. *Assume* $\Xi \vdash C$. *If* $\Xi \vdash C \xRightarrow{t}$, *then* $\Xi \vdash \varphi_t : wf$ *(where* φ_t *is the trace* t *interpreted to conform to Tab. 6, i.e., the names of* t *are replaced by variables).*

Proof. By straightforward induction. The proof works similar to the proof in [27], which shows that the behavior of a component is a legal trace. Note in this context that φ_t is of a restricted form: it is constructed by prefixing and the empty trace, only. □

$$\frac{\Xi \vdash C \Longrightarrow \Xi \vdash \acute{C}}{\Xi \vdash C \parallel \varphi \rightarrow \Xi \vdash \acute{C} \parallel \varphi}\ \textsc{Par-Int} \qquad \frac{\vdash a \lesssim_\sigma b \qquad \Xi_1 \vdash C \xrightarrow{a} \acute{\Xi}_1 \vdash \acute{C} \qquad \Xi_1 \vdash \varphi \xrightarrow{b} \acute{\Xi}_2 \vdash \acute{\varphi}}{\Xi_1 \vdash C \parallel \varphi \rightarrow \acute{\Xi}_1 \vdash \acute{C} \parallel \acute{\varphi}\sigma}\ \textsc{Par}$$

$$\frac{\Xi \vdash \varphi : wf^?}{\Xi \vdash \nu(\Xi').(C \parallel n\langle \mathsf{let}\, x{:}T = o.l(\vec{v})\, \mathsf{in}\, t\rangle \parallel \varphi) \rightarrow \lightning}\ \textsc{Par-Err-Call}$$

$$\frac{\Xi \vdash \varphi : wf^?}{\Xi \vdash \nu(\Xi').(C \parallel n\langle v\rangle \parallel \varphi) \rightarrow \lightning}\ \textsc{Par-Err-Ret}$$

Table 10. Parallel composition

4 Scheduling and Asynchronous Testing of Creol Objects

Next we put together the (external) behavior of an object (Sect. 2) and its intended behavior specified as in Sect. 3. Table 10 defines the interaction of the interface description with the component, basically by synchronous parallel composition. Both φ and the component must engage in corresponding steps, which, for incoming communication schedules the order of interactions with the component whereas for outgoing communication the interaction will take place only if it matches an outgoing label in the specification and an error is raised if input is required by the specification. The component can proceed on its own via internal steps (cf. rule Par-Int). Rule Par requires that, in order to proceed, the component and the specification must engage in the "same" step, where φ's step b is matched against the step a of the component. The matching is not simple pattern matching as it needs to take into account in particular the two different kinds of bindings in the specification language, $\nu(x{:}T)$ as the freshness assertion and $(x{:}T)$ representing standard variable declarations; see Def. 3 below. The rules Par-Err-Call and Par-Err-Ret report an error if the specification requires an input as the next step and the object however could do an output, either a call or a return. In the rule $\lightning$ indicates the occurrence of an error. Note that the equivalence relation, according to the rule Eq-Switch, allows the reordering of outputs, but not of inputs.

Definition 3 (Matching). *Given two labels a_1 and a_2, we write $\vdash a_1 \lesssim_\sigma a_2$ (read "a_1 matches a_2 with substitution σ"), if that judgment can be derived by the rules of Tab. 11.*

The rules of Tab. 11 work as follows. They define the matching relation between two labels (written $\vdash a_1 \lesssim_\sigma a_2$), where a_1 is the label produced by the component and a_2 the one specified by the interface description. The subscript σ is the substitution, a mapping from variables to names, that gives rise to the match.

$$\frac{}{\vdash () \lesssim () : ok}\ \text{M-Empty} \qquad \frac{\vdash \Xi_1 \lesssim \Xi_2 : ok}{\vdash \nu(n{:}T), \Xi_1 \lesssim \nu(n{:}T), \Xi_2 : ok}\ \text{M-NDec}$$

$$\frac{\vdash \Xi_1 \lesssim \Xi_2 : ok}{\vdash \nu(n{:}T), \Xi_1 \lesssim (n{:}T), \Xi_2 : ok}\ \text{M-Dec}_1 \qquad \frac{\vdash \Xi_1 \lesssim \Xi_2 : ok}{\vdash \Xi_1 \lesssim (n{:}T), \Xi_2 : ok}\ \text{M-Dec}_2$$

$$\frac{\vdash a_1 \lesssim_\sigma a_2 : ok \qquad \vdash \Xi_1 \lesssim \Xi_2\sigma : ok}{\vdash \Xi_1.a_1 \lesssim_\sigma \Xi_2.a_2 : ok}\ \text{M-Lab}$$

Table 11. Matching of labels

The difference between the two syntactic categories therefore is that a_2 may contain variables and a_1 may not, and furthermore, the grammar for a_2 allows bindings of the form $(x{:}T)$ and $\nu(x{:}T)$, whereas the first variant does not occur for labels a_1. A label a (for both cases) consists of a binding part and the core of the label without bindings. In slight abuse of notation, we write Ξ for the binding part or context. In rule M-Lab for matching the two labels, Ξ_1 thus contains bindings of the form $\nu(n{:}T)$, i.e., from names to types, denoting the *new* names exchanged with the object in that step. Ξ_2's bindings on the other hand, associating variables with types, are of the form $(x{:}T)$ or $\nu(x{:}T)$. The label $\Xi_1.a_1$ is matched against $\Xi_2.a_2$, as given by M-Lab in two steps. First, the cores a_1 and a_2 of the two labels are matched against each other by standard pattern matching, written $\vdash a_1 \lesssim_\sigma a_2$. In other words, $a_1 = a_2\sigma$, where σ is the (uniquely determined) substitution, which, when applied to a_2, gives a_1.[5]

The outer binding parts Ξ_1 and Ξ_2 are checked afterwards, as specified in the remaining 4 rules, where the variables of Ξ_2 are replaced by names, as given by the matching substitution σ. Note that the well-formedness conditions assure, that $\Xi_2\sigma$ no longer contains variable bindings, only bindings from names to types. Note further that we consider the contexts Ξ_1 and Ξ_2 as un-ordered, i.e., writing e.g. $(o{:}T), \Xi$ does not indicate that the binding $(o{:}T)$ occurs left-most in Ξ. In other words, the contexts Ξ, as usual, are understood up-to re-ordering. Rule M-NDec stipulates that if the specification requires a new name, the transmitted name must indeed be fresh. If, however, the specification just introduces a variable without insisting on freshness, then either the name can be fresh (cf. rule M-Dec$_1$) or the name had already been introduced, in which case it is ignored (cf. rules M-Dec$_2$). Finally, two empty contexts clearly match (cf. rule M-Empty).

Example 2. To illustrate the testing we sketch the well-known example of a travel agency. A client asks the travel agent for a cheap flight and the travel agent finds

[5] As an aside: assuming that both labels are checked for well-formedness, a type-mismatch between the name and the variable does not occur.

the cheapest flight by asking the flight companies. To test an implementation of the travel agent program we give a specification modeling the behavior of the client and the flight companies and specifying the expected behavior of the travel agent. The client sends two messages. First a start message and then the request. The travel agent tries to get the price information from the flight companies and then reports the result to the client.

$$\begin{aligned}\varphi_b = {} & n_{c1}\langle \mathit{call}\ b.\mathit{start}()\rangle? \,.\, n_{c1}\langle \mathit{return}()\rangle! \,.\, n_{c2}\langle \mathit{call}\ b.\mathit{getPrice}(x)\rangle? \,. \\ & n_1\langle \mathit{call}\ p_1.l(x)\rangle! \,.\, n_2\langle \mathit{call}\ p_2.l(x)\rangle! \,. \\ & n_1\langle \mathit{return}(v_1)\rangle? \,.\, n_2\langle \mathit{return}(v_2)\rangle? \,.\, n_{c2}\langle \mathit{return}(\mathit{min}_v)\rangle!\end{aligned}$$

5 Implementing a Specification-driven Creol Interpreter

The operational semantics of the object-oriented language Creol [20] is formalized in rewriting logic [22] and executable on the Maude rewriting engine [9]. To obtain a *specification-driven interpreter* for testing Creol objects, we have formalized our behavioral interface specification language in rewriting logic, too. In the combined implementation we synchronize communication between specification terms and objects. The specification generates the required input to the object and tests whether the output behaviour of the object conforms to the specification. The original Creol interpreter consists of 21 rewrite rules and the extension adds 20 more.

We have argued that specified method calls should not be placed into the callee's input queue, but the call should be answered immediately. I.e., if an incoming call is specified and the lock of the object is free, the corresponding method code should start executing immediately. In the current version of the interpreter the incoming messages are generated from the specification, which amounts to the same as only allowing scheduled calls to interact with the object.

A Creol state configuration is a multiset of objects, classes, and messages. The rewrite rules for state transitions are on the form `rl Cfg => Cfg'`, effectively evolving the state of one object by executing a statement. Some statements generate new messages. Finally, some rules are concerned with scheduling processes and receiving messages. For the scheduling interpreter we introduce terms `Spec` for specifications and add rules on the form `(Spec || O) Cfg => (Spec' || O') Cfg'` to test the object `O` with respect to `Spec`, where `||` represents the synchronous parallel composition. Each rule evolves the state of a specification and the state of an object in a synchronized manner: any interaction only takes place when it matches a complementary label in the specification. For example, the Par rule in Tab. 10 is implemented by several Maude rules, of which we show `Par-incoming-call` and `Par-remote-async-call`, that handle the cases of synchronized incoming and outgoing calls; we also show the `Par-Err-Call` rule in Tab. 12. The rules are conditional rewrite rules, in which conditions of the form `Var:=term` bind `term` to the variable `Var`. Parts of the term that are not changed, like attributes, are represented by "...".

```
crl <call(T,R,M,P)?.sp>(O) || <O:C | ..., Pr:idle, ...> Cfg
 => <app(getS(Res),sp)>(O) || <O:C | ..., Pr:synch, ...> getM(Res) Cfg
if Res:=procLab(O , call(T,R,M,P)?) [label Par-incoming-call] .

crl <call(T,R,M,P)!.sp>(O) || <O:C | ...,Pr:{L | call(A;E;Q;EL);SL},...> Cfg
 => <app(Sub,sp)>(O)       || <O:C | ...,Pr:{insert(A,Lab,L) | SL},...> Cfg
    (invoc(O,Lab,Q,Args) from O to Rcv)
if Lab:=label(O,N) ∧ Rcv:=evalGuard(E,(S::L),noMsg) ∧
   Args:=evalGuardList(EL,(S::L),noMsg) ∧
   Sub:=matchCall(Lab,Rcv,Q,Args,call(T,R,M,P)) ∧ noMismatch(Sub)
[label Par-remote-async-call] .

crl <inSp>(O)    || <O:C | ...,Pr:{L | call(A;E;Q;EL);SL},...> Cfg
 => <epsilon>(O) || <O:C | ...,Pr:{L | call(A;E;Q;EL);SL},...> Cfg
    errorMsg("ERROR") if E=/="this" [label Par-Err-Call] .
```

Table 12. Sample Maude rules

The rule `Par-incoming-call` combines the R-PREF rule in Tab. 9 for the specification with the CALLI rule in Tab. 5 for interface behavior via the PAR rule. The rule only applies if the process, `Pr` of the object `<O:C | ...>` is `idle` (i.e., the lock is free). The specification for `O`, `<call(T,R,M,P)? . sp>(O)`, starts with an incoming call label with thread name `T`, receiver `R`, method name `M`, and parameters `P`, and could by R-PREF reduce to `sp`. The careful reader might expect that the receiver mentioned in the specification should be the same as the object identifier `O`. However since a specification can contain variables, the receiver `R` *might* be identical to `O` but it may also be a variable, which will be matched with `O` in the `procLab` function. The function `procLab` (process label) generates concrete values from the variables in the specification label; builds an *invoc* message, i.e. a term representing a method call; and returns the message and a mapping of the variables to the values. The message and the substitution are extracted by the functions `getM` and `getS`, respectively. The message is placed into the configuration and the substitution is applied to `sp` using the `app` function. Method binding and the rules for executing the bound code are specified by equations in the Creol interpreter. Since equations will be applied before any other rewrite rules this ensures that the execution of the code resulting from the call starts before any other invoc message can interfere.

In the `Par-remote-async-call` rule the object is in a state where the next step in the executing process is an outgoing call (indicated by the statement `call(A;E;Q;EL)`) and the specification starts with a call out. The `matchCall` function tries to match the concrete values derived from the object's state (the thread name `Lab`, which is a combination of the object name and an internal counter in the object, the receiver `Rcv`, and the arguments `Args`) against the variables in the label. The condition `noMismatch(Sub)` blocks the conditional rule if no match is possible, otherwise the outgoing call takes place and the

substitution `Sub` is applied to the remainder of the specification. The last rule implements the Par-Err-Call rule. The distinction between input and output specifications is enforced by different subsorts: the variable `inSp` matches all specifications of incoming messages. When the next step of the executing process is a call statement, then this leads to an error, as expected.

In the implementation we focus on the run-time behavior of specifications which implies that well-formedness checking of specifications is not considered, we assume that specifications are well-formed. We have not yet implemented the ν-binder and thus, we do not check in the matching for freshness of names but treat all variables as standard variables. (Cf. Def. 3)

6 Conclusion

We have presented a formalization of the interface behavior of Creol together with a behavioral interface specification language. We have formally described how to use this specification language for black-box testing of asynchronously communicating Creol components and we have presented our rewriting logic implementation of the testing framework.

Related work Systematic testing is indispensable to assure quality of software and systems (cf. [24,25,15,5,4], amongst others). [8] presents an approach to integrate black-box and white-box testing for object-oriented programs. Equivalence is based on the idea of observably equivalent terms and fundamental pairs as test cases, but not in an asynchronous setting (and as in [3] [13] [12] [14]). In the approach, pairs of (ground) terms are used for the test cases. Testing for *concurrent* object-oriented programs based on synchronization sequences is investigated in [7], using Petri nets and OBJ as foundation. Long in his thesis [21] presents ConAn ("concurrency analyser"), which generates test drivers from test scripts. The method allows to specify sequences of component method calls and the order in which the calls should be issued. It can be seen as an extension of the testing method for monitors from [6]. For scheduling the intended order, an external *clock* is used, which is introduced for the purpose of testing, only. In the context of $C^{\#}$, [17] presents model-based analysis and model-based testing, where abstract models of object-oriented programs are tested. The approach, however, does not target concurrent programs.

Even if not specifically targeting Creol, [19] pursues similar goals as this paper, validating component interfaces specified in rewriting logic. In contrast to here, the interface behavior is specified by first-order logic over traces, where from a given predicate an assumption part and a guarantee part can be derived. The assumption part of the specification is used to generate arbitrary input to the component under test, while the guarantee part is used for testing that the output from the component conforms to the given predicate. Our approach is more specific in that we schedule incoming calls to a component, and test the output behavior. Our specification language is not first-order logic but a recursive language over communication labels.In [26], the authors target Creol as language

and investigate how different schedulings of object activity restrict the behavior of a Creol object, thus leading to more specific test scenarios. The focus, however, is on the *intra*-object scheduling, and the test purposes are given as assertions on the *internal* state of the object. This is in contrast to the setting here, focusing on the interface communication. The testing methodologies are likewise different. We execute the behavioral trace specification directly in composition with the implementation being tested. They use a scheduling strategy and a model for an object implementation to generate test cases which then are used afterwards to test for compliance with an implemented Creol object.

Future work We plan to extend the theory to components under test instead of single objects. This leads to complex scheduling policies and complex specifications. Furthermore, there are several interesting features of the Creol language which may be added, including first-class futures, promises, processor release points, inheritance and dynamic class updates. For the specification language we want to investigate how to extend it with assertion statements on labels, which leads to scheduling policies sensitive to the *values* in the communication labels. Natural further steps for the implementation are to extend it to include a check for well-formedness according to Tab. 7, and also to modify the matching algorithm to distinguish between fresh and already known namesas specified in Tab. 11 . The generation of Creol messages from specifications can also be made more sophisticated to achieve better test coverage. It is also interesting to combine the approach we describe here with model checking and abstraction. By using the built-in *search* functionality of Maude, model checking of invariants can be done easily. We plan to additionally use Maude's LTL model checker with our testing framework.

Acknowledgement We thank Andreas Grüner for giving insight to the field of testing of (concurrent) object-oriented languages, the members of the PMA group for valuable feedback and the anonymous referees for insightful and constructive criticism.

References

1. M. Abadi and L. Cardelli. *A Theory of Objects*. Monographs in Computer Science. Springer, 1996.
2. E. Ábrahám, I. Grabe, A. Grüner, and M. Steffen. Behavioral interface description of an object-oriented language with futures and promises. 78(7), 2009. Special issue of the with selected contributions of NWPT'07. The paper is a reworked version of an earlier UiO Technical Report TR-364, Oct. 2007.
3. G. Bernot, M.-C. Gaudel, and B. Marre. Software testing based on formal specifications. *IEEE Software Engineering Journal*, 6(6):387–405, Nov. 1991.
4. A. Bertolino. Software testing research: Achievements, challenges, dreams. In *FOSE 2007: Future of Software Engineering*, pages 85–103. IEEE Computer Society Press, 2007.

5. R. V. Binder. *Testing Object-Oriented Systems, Models, Patterns, and Tools*. Addison-Wesley, 2000.
6. P. Brinch Hansen. Reproducible testing of monitors. *Software – Practice and Experience*, 8(223–245), 1978.
7. H. Y. Chen, Y. X. Sun, and T. H. Tse. A strategy for selecting synchronization sequences to test concurrent object-oriented software. In *Proceedings of the 27th International Computer Software and Application Conference (COMPSAC 2003)*. IEEE Computer Society Press, 2003.
8. H. Y. Chen, T. H. Tse, F. T. Chan, and T. Y. Chen. In black and white: An integrated approach to class-level testing of object-oriented program. *ACM Transactions of Software Engineering and Methodology*, 7(3):250–295, 1998.
9. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. The Maude 2.0 system. In R. Nieuwenhuis, editor, *RTA 2003*, volume 2706 of *LNCS*, pages 76–87. Springer, 2003.
10. The Creol language. `http://heim.ifi.uio.no/creol`.
11. W.-P. de Roever, F. S. de Boer, U. Hannemann, J. Hooman, Y. Lakhnech, M. Poel, and J. Zwiers. *Concurrency Verification: Introduction to Compositional and Noncompositional Proof Methods*. Cambridge University Press, 2001.
12. R.-K. Doong and P. G. Frankl. Case studies on testing object-oriented programs. In *TAV4: Proceedings of the symposium on Testing, analysis, and verification*, pages 165–177. ACM Press, 1991.
13. R.-K. Doong and P. G. Frankl. The ASTOOT approach to testing object-oriented programs. *ACM Transactions on Software Engineering and Methodology*, 3(2):101–130, 1994.
14. P. G. Frankl and R.-K. Doong. Tools for testing object-oriented programs. In *Proceedings of the 8th Pacific Northwest Conference on Software Quality*, pages 309–324, 1990.
15. M.-C. Gaudel. Testing can be formal, too. In P. D. Mosses, M. Nielsen, and M. I. Schwarzbach, editors, *TAPSOFT 1995*, volume 915 of *LNCS*, pages 82–96. Springer, 1995.
16. K. Honda. Types for dyadic interaction. pages 509–523.
17. J. Jacky, M. Veanes, C. Campbell, and W. Schulte. *Model-Based Software Testing and Analysis with C#*. Cambridge University Press, 2008.
18. A. Jeffrey and J. Rathke. A fully abstract may testing semantics for concurrent objects. In *17th Annual IEEE Symposium on Logic in Computer Science*, pages 101–112. IEEE Computer Society Press, 2002.
19. E. B. Johnsen, O. Owe, and A. B. Torjusen. Validating behavioral component interfaces in rewriting logic. *Fundamenta Informaticae*, 82(4):341–359, 2008.
20. E. B. Johnsen, O. Owe, and I. C. Yu. Creol: A type-safe object-oriented model for distributed concurrent systems. *Theoretical Computer Science*, 365(1–2):23–66, Nov. 2006.
21. B. Long. *Testing Concurrent Java Components*. PhD thesis, University of Queensland, July 2005.
22. J. Meseguer. Conditional rewriting as a unified model of concurrency. *Theoretical Computer Science*, 96:73–155, 1992.
23. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, part I/II. *Information and Computation*, 100:1–77, Sept. 1992.
24. G. J. Myers. *The Art of Software-Testing*. John Wiley & Sons, New York, 1979.
25. R. Patton. *Software Testing*. SAMS, 2 edition, July 2005.

$$\frac{}{\Delta \vdash \mathbf{0} : ()}\ \text{T-Empty} \qquad \frac{\Delta, \Theta_2 \vdash C_1 : \Theta_1 \quad \Delta, \Theta_1 \vdash C_2 : \Theta_2}{\Delta \vdash C_1 \parallel C_2 : \Theta_1, \Theta_2}\ \text{T-Par} \qquad \frac{\Delta \vdash C : \Theta, n{:}T}{\Delta \vdash \nu(n{:}T).C : \Theta}\ \text{T-Nu}$$

$$\frac{;\, \Delta, c{:}T \vdash [\![O]\!] : T}{\Delta \vdash c[\![O]\!] : (c{:}T)}\ \text{T-NClass} \qquad \frac{;\, \Delta \vdash c : [\![T_F, T_M]\!] \quad ;\, \Delta, o{:}c \vdash [F] : [T_F]}{\Delta \vdash o[c, F, L] : (o{:}c)}\ \text{T-NObj}$$

$$\frac{;\, \Delta, n{:}[T] \vdash t : T}{\Delta \vdash n\langle t \rangle : (n{:}[T])}\ \text{T-NThread} \qquad \frac{\Delta' \leq \Delta \quad \Theta \leq \Theta' \quad \Delta \vdash C : \Theta}{\Delta' \vdash C : \Theta'}\ \text{T-Sub}$$

Table 13. Typechecking (1)

26. R. Schlatte, B. Aichernig, F. de Boer, A. Griesmayer, and E. B. Johnsen. Testing concurrent objects with application-specific schedulers. In J. Fitzgerald and A. Haxthausen, editors, *ICTAC 2008*, volume 5160 of *LNCS*. Springer, 2008.
27. M. Steffen. *Object-Connectivity and Observability for Class-Based, Object-Oriented Languages*. Habilitation thesis, Technische Faktultät der Christian-Albrechts-Universität zu Kiel, July 2006.
28. K. Takeuchi, K. Honda, and M. Kubo. An interaction-based language and its typing system. pages 398–413.

A Appendix

A.1 Type checking

Type checking is split into two levels, one on the level of components (cf. Tab. 13) and one on the level of thread, expressions, and their sub-phrases (cf. Tab. 14). For components, the rules formalize a judgement of the form $\Delta \vdash C : \Theta$, where Δ is the assumption context and Θ the commitment context. Both associate (class, thread, and object) names with their respective types, where the assumption context takes care of those names which are part of the environmemt, whereas dually Θ is reponsible for the names of the componenent. At the level of threads and expressions, the judgments are of the form $\Delta; \Gamma \vdash t : T$, where Δ is the (assumption) name context, and Γ contains the bindings of the local variables.

A.2 Structural congruence

Components are considered up-to a standard structural congruence, which is formalized in Tab. 15. The rule for scope extrusion on the left-bottom is applied under the side-condition that n does not occur free in C_1. The congruence relation is imported into the reduction relation via the rules of Tab. 16.

A.3 Traces

A trace of a well-typed component is a sequence of external steps; we write $\Xi_1 \vdash C_1 \stackrel{t}{\Longrightarrow} \Xi_2 \vdash C_2$ when the component $\Xi_1 \vdash C_1$ evolves to $\Xi_2 \vdash C_2$

by executing the trace t. The corresponding rules are given in Tab. 17. For $\Xi_1 \vdash C_1 \stackrel{\epsilon}{\Longrightarrow} \Xi_2 \vdash C_2$, we write shorter $\Xi_1 \vdash C_1 \Longrightarrow \Xi_2 \vdash C_2$, where ϵ denote the empty trace.

$$\frac{\Gamma;\Delta \vdash c : [\![l_1{:}U_1,\ldots,l_k{:}U_k]\!] \quad \Gamma;\Delta \vdash m_i : U_i \quad m_i = \varsigma(s_i{:}c).\lambda(\vec{x}_i{:}\vec{T}_i).t_i}{\Gamma;\Delta \vdash [\![l_1 = m_1,\ldots,l_k = m_k]\!] : c}\ \text{T-Class}$$

$$\frac{\Gamma;\Delta \vdash c : [\![l_1{:}U_1,\ldots,l_k{:}U_k]\!] \qquad \Gamma;\Delta \vdash f_i : U_i \qquad f_i = \varsigma(s_i{:}c).\lambda().v_\perp}{\Gamma;\Delta \vdash [l_1 = f_1,\ldots,l_k = f_k] : c}\ \text{T-Obj}$$

$$\frac{\Gamma,\vec{x}{:}\vec{T};\Delta,s{:}c \vdash t : T' :: \acute{\Gamma};\acute{\Delta} \quad \Gamma;\Delta \vdash c : T \quad T = [\![\ldots,l{:}\vec{T} \to T',\ldots]\!]}{\Gamma;\Delta \vdash \varsigma(s{:}c).\lambda(\vec{x}{:}\vec{T}).t : T.l}\ \text{T-Memb}$$

$$\frac{\Gamma;\Delta,s{:}c \vdash c : [\![\ldots,l : \mathsf{Unit} \to c',\ldots]\!]}{\Gamma;\Delta \vdash \varsigma(s{:}c).\lambda().\bot_{c'} : c'}\ \text{T-Undef}$$

$$\frac{\Gamma;\Delta \vdash v : c \quad \Gamma;\Delta \vdash c : T \quad \Gamma;\Delta \vdash v' : T.l}{\Gamma;\Delta \vdash v.l := v' : c}\ \text{T-FUpdate} \qquad \frac{\Gamma;\Delta \vdash c : [\![T]\!]}{\Gamma;\Delta \vdash \mathsf{new}\ c : c}\ \text{T-NewC}$$

$$\frac{\Gamma;\Delta \vdash e : T_1 \qquad \Gamma,x{:}T_1;\Delta \vdash t : T_2}{\Gamma;\Delta \vdash \mathsf{let}\ x{:}T_1 = e\ \mathsf{in}\ t : T_2}\ \text{T-Let}$$

$$\frac{\Gamma;\Delta \vdash v_1 : T_1 \quad \Gamma;\Delta \vdash v_2 : T_1 \quad \Gamma;\Delta \vdash e_1 : T_2 \quad \Gamma;\Delta \vdash e_2 : T_2}{\Gamma;\Delta \vdash \mathsf{if}\ v_1 = v_2\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : T_2}\ \text{T-Cond}$$

$$\frac{\Gamma;\Delta \vdash v : c \quad \Gamma;\Delta \vdash c : [\![\ldots,l{:}\mathsf{Unit} \to T,\ldots]\!] \quad \Gamma;\Delta \vdash e_1 : T_2 \quad \Gamma;\Delta \vdash e_2 : T_2}{\Gamma;\Delta \vdash \mathsf{if}\ \mathsf{undef}(v.l())\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : T_2}\ \text{T-Cond}_\perp$$

$$\frac{}{\Gamma;\Delta \vdash \mathsf{stop} : T}\ \text{T-Stop} \qquad \frac{}{\Gamma;\Delta \vdash () : \mathsf{Unit}}\ \text{T-Unit}$$

$$\frac{\Gamma;\Delta \vdash v : c \quad \Gamma;\Delta \vdash c : [\![\ldots,l{:}\vec{T} \to T,\ldots]\!] \quad \Gamma;\Delta \vdash \vec{v} : \vec{T}}{\Gamma;\Delta \vdash v@l(\vec{v}) : [T]}\ \text{T-CallA}$$

$$\frac{\Gamma;\Delta \vdash n : [T] \qquad \Gamma;\Delta \vdash o{:}c}{\Gamma;\Delta \vdash \mathsf{claim}@(n,o) : T}\ \text{T-Claim} \qquad \frac{\Gamma;\Delta \vdash n : [T]}{\Gamma;\Delta \vdash \mathsf{get}@n : T}\ \text{T-Get}$$

$$\frac{\Gamma(x) = T}{\Gamma;\Delta \vdash x : T}\ \text{T-Var} \qquad \frac{\Delta(x) = T}{\Gamma;\Delta \vdash n : T}\ \text{T-Name}$$

$$\frac{\Delta \vdash o : c}{\Gamma;\Delta \vdash \mathsf{suspend}(o) : \mathsf{Unit}}\ \text{T-Suspend} \qquad \frac{\Delta \vdash o : c}{\Gamma;\Delta \vdash \mathsf{grab}(o) : \mathsf{Unit}}\ \text{T-Grab}$$

$$\frac{\Delta \vdash o : c}{\Gamma;\Delta \vdash \mathsf{release}(o) : \mathsf{Unit}}\ \text{T-Release}$$

Table 14. Typechecking (2)

$$\mathbf{0} \parallel C \equiv C \qquad C_1 \parallel C_2 \equiv C_2 \parallel C_1 \qquad (C_1 \parallel C_2) \parallel C_3 \equiv C_1 \parallel (C_2 \parallel C_3)$$
$$C_1 \parallel \nu(n{:}T).C_2 \equiv \nu(n{:}T).(C_1 \parallel C_2) \qquad \nu(n_1{:}T_1).\nu(n_2{:}T_2).C \equiv \nu(n_2{:}T_2).\nu(n_1{:}T_1).C$$

Table 15. Structural congruence

$$\frac{C \equiv \rightsquigarrow \equiv C'}{C \rightsquigarrow C'} \qquad \frac{C \rightsquigarrow C'}{C \parallel C'' \rightsquigarrow C' \parallel C''} \qquad \frac{C \rightsquigarrow C'}{\nu(n{:}T).C \rightsquigarrow \nu(n{:}T).C'}$$
$$\frac{C \equiv \xrightarrow{\tau} \equiv C'}{C \xrightarrow{\tau} C'} \qquad \frac{C \xrightarrow{\tau} C'}{C \parallel C'' \xrightarrow{\tau} C' \parallel C''} \qquad \frac{C \xrightarrow{\tau} C'}{\nu(n{:}T).C \xrightarrow{\tau} \nu(n{:}T).C'}$$

Table 16. Reduction modulo congruence

$$\frac{C_1 \Longrightarrow C_2}{\Xi_1 \vdash C_1 \overset{\epsilon}{\Longrightarrow} \Xi_2 \vdash C_2}\ \textsc{Internal} \qquad \frac{\Xi_1 \vdash C_1 \xrightarrow{a} \Xi_2 \vdash C_2}{\Xi_1 \vdash C_1 \overset{a}{\Longrightarrow} \Xi_2 \vdash C_2}\ \textsc{Base}$$
$$\frac{\Xi_1 \vdash C_1 \overset{t_1}{\Longrightarrow} \Xi_2 \vdash C_2 \qquad \Xi_2 \vdash C_2 \overset{t_2}{\Longrightarrow} \Xi_3 \vdash C_3}{\Xi_1 \vdash C_1 \overset{t_1 t_2}{\Longrightarrow} \Xi_3 \vdash C_3}\ \textsc{Conc}$$

Table 17. Traces

Chapter 12

Paper #5: Model testing asynchronously communicating objects using modulo AC rewriting

Olaf Owe, Martin Steffen, and Arild B. Torjusen

In *Proceedings of Model-Based Testing MBT'10 (ETAPS Satellite Workshop), March 2010.* To appear in *Electronic Notes in Theoretical Computer Science*

Model Testing Asynchronously Communicating Objects using Modulo AC Rewriting

Olaf Owe, Martin Steffen, and Arild B. Torjusen

Department of Computer Science, University of Oslo, Norway

Abstract

Testing and verification of asynchronously communicating objects in open environments are challenging due to non-determinism. We explore a formal approach for black-box testing by proposing an interface specification language that gives an assumption-commitment style description of an object's behavior. The approach is applied to Creol objects. Creol is a high-level, object-oriented modelling language, hence we do model-based testing of behavioral models. The testing is done by synchronising execution of a specification and the component under test. Due to the asynchronous nature of communication, testing should be done up-to observational equivalence. This leads to a large increase in the reachable state space for the test cases. We reduce the state space by using facilities for rewriting modulo AC (associativity and commutativity) built into the rewriting logic system Maude, and explore the state space by breadth first search. We present experimental results that show the usefulness of this approach.

Keywords: Testing and verification, asynchronous method calls, active objects, rewriting logic, formal semantics.

1 Introduction

Systematic testing is indispensable to assure reliability and quality of software and systems. Hosts of different testing approaches and frameworks have been proposed and put to (good) use over the years. Formal methods and program language theory have proven valuable to render testing practice a more formal, systematic discipline (cf. e.g. [18,3]). Formal approaches to testing have gained momentum in recent years, as for instance witnessed by the trend towards model-based testing [13,5]. In previous work [22] we presented a formal approach for black-box specification-based testing of asynchronously communicating components in open environments together with an implementation of a testing framework. In this paper we show how to extend the approach to *verification* of components and present experimental results that show the usefulness of our approach.

⋆ Part of this work has been supported by the EU-project IST-33826 *Credo: Modeling and analysis of evolutionary structures for distributed services*, *HATS: Highly Adaptable and Trustworthy Software using Formal Methods* (`http://www.hats-project.eu`), and the German-Norwegian DAAD-NWO exchange project *Avabi* (Automated validation for behavioral interfaces of asynchronous active objects).

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: `www.elsevier.nl/locate/entcs`

We do this in the context of Creol [12,31], a high-level, object-oriented modelling language for distributed systems. Object-orientation is a natural choice, as object modelling is the fundamental approach to open distributed systems as recommended by RM-ODP [27]. For such systems an *asynchronous* communication model is advantageous as it decouples caller and callee thus avoiding unnecessary waiting for method returns. On the downside, asynchronicity makes verifying and testing models more challenging. In an asynchronous system, communication delays due to the network or to queuing may lead to message overtaking and the resulting non-determinism leads to a state space explosion.

It is generally accepted that the way to tackle complex systems is to "divide-and-conquer", i.e., consider components interacting with their environment. Abstracting from internal executions, the black-box behavior of Creol components is given by interactions at their *interface*. We use a concise language over communication labels to specify components and the expected behavior of a component is given as a set of traces at the interface. Both input and output interactions are specified but play quite different roles. As input events are not under the control of the object, but of the environment, input is considered as assumptions about the environment whereas output describes commitments of the object. This separation of concerns between interaction under the control of the component and coming from the environment leads to an assumption-commitment style specification of a component's behavior by defining the valid observable output behavior, assuming a certain scheduling of the input.

For input interactions, we ensure that the specified assumptions on the environment are fulfilled by *scheduling* the incoming calls in the order specified, while for output events, which are controlled by the component, we *test* that the events occur as specified. Scheduling and testing of a component are done by synchronizing the component's execution with the specification. As a result, the scheduling is enforced in the execution of the component and the actual outgoing interactions from the component are tested against the output events in the specification. This gives a framework for testing whether an implementation of a component conforms with the interface specification. Incorrect or nonconforming behavior of the component under a given scheduling is reported as an error by the testing framework.

Due to message delays and overtaking, the order in which outgoing messages from a component are observed by an external observer does not necessarily reflect the order in which they were actually sent. Testing is based on behavior observable at the interface, and the order of outgoing communication should therefore not affect the test results. The operational semantics of the specification language takes the asynchronous nature of the communication model into consideration by treating certain reorderings of output events as observationally equivalent, and testing is done up-to *observational equivalence*.

Reordering of output events can be expressed by defining sequences of output events as *associative* and *commutative*. We argue that our testing framework is especially well suited to implement this since, using the rewriting logic system Maude, associativity and commutativity can be declared using *equational attributes* [10] which allows efficient evaluation of such specifications.

This paper extends [22] which introduced and gave the formal basis for the

approach to testing that we explore further here, the main contributions are:

Verification We provide an implementation in the rewriter Maude and use Maude's *search* functionality for state exploration (for rewriting modulo AC) for verification of components and investigate how the support for AC reasoning built in into Maude contributes to state space reduction in verification of asynchronously communicating components.

Experimental results We present *experimental results* from using the Maude rewriting tool which give empirical evidence of the benefits of our method. We compare, in two series of experiments, the influence on the state space of using Maude's built in AC support against explicit representation of all possible reorderings of output events (with the same semantics). Using AC rewriting may considerably reduce the resource consumption when testing asynchronously communicating objects. AC rewriting significantly pays off in terms of time and the number of rewrites.

We review the formalisation of Creol in Sect. 2, where some of the technicalities from the previous paper are repeated when necessary. The corresponding behavioral interface specification language and an explanation of how this is used for testing are given in Sect. 3. In Sect. 4, we describe the executable implementation of the theory. The *experimental results* are in Sect. 5.

2 The Creol modeling language

We formalise Creol, a high-level, object-oriented modelling language for distributed systems, Creol features active objects and asynchronous method calls.

In contrast with object-oriented languages based on multi-threading, such as *Java* or $C^{\#}$, the language features *active objects*. The unit of activity is the object; every process belongs to an object, and activity does not cross object borders. Communication is based on exchanging messages *asynchronously*, and is *asymmetric* in the sense that there are linguistic means to *send* a message, but not to *accept* a message: objects are always input-enabled. On the callee side of a method call therefore each object possesses an input "queue" in which incoming messages are waiting to be served by the object. To avoid uncontrolled interference, each object acts as a *monitor*; at most one method body is executing at each point in time. By default the choice of which method call in the input queue that enters the object next is *non-deterministic*.

After the abstract syntax, we sketch the operational semantics, concentrating on the external behavior, i.e., the message exchange with the environment.

2.1 Syntax

The abstract syntax, in the style of standard object calculi, is given in Tab. 1. Names n represent references to classes, to objects, and to threads. To facilitate reading, we allow ourselves to write o and its syntactic variants for names referring to objects, c for classes, and n when being unspecific. A *component* C is a collection of classes, objects, and (named) threads, with $\mathbf{0}$ representing the empty component. The sub-entities of a component are composed using the parallel-construct $\|$. The

$$
\begin{array}{rcll}
C & ::= & \mathbf{0} \mid C \parallel C \mid \nu(n{:}T).C \mid c[\![F, M]\!] \mid o[c, F, L] \mid n\langle t\rangle & \text{component} \\
F & ::= & l = f, \ldots, l = f & \text{fields} \\
M & ::= & l = m, \ldots, l = m & \text{method suite} \\
m & ::= & \varsigma(n{:}T).\lambda(x{:}T, \ldots, x{:}T).t & \text{method} \\
f & ::= & \varsigma(n{:}T).\lambda().v \mid \varsigma(n{:}T).\lambda().\bot_{n'} & \text{field} \\
t & ::= & v \mid \mathsf{stop} \mid \mathsf{let}\, x{:}T = e\, \mathsf{in}\, t & \text{thread} \\
e & ::= & t \mid \mathsf{if}\, v = v\, \mathsf{then}\, e\, \mathsf{else}\, e \mid \mathsf{if}\, \mathit{undef}(v.l())\, \mathsf{then}\, e\, \mathsf{else}\, e & \text{expr.} \\
 & \mid & v@l(\boldsymbol{v}) \mid v.l(\boldsymbol{v}) \mid v.l() \mid v.l := \varsigma(s{:}T).\lambda().v & \\
 & \mid & \mathsf{new}\, n \mid \mathsf{claim}@(n, n) \mid \mathsf{get}@n \mid \mathsf{suspend}(n) \mid \mathsf{grab}(n) \mid \mathsf{release}(n) & \\
v & ::= & x \mid n \mid () & \text{values} \\
L & ::= & \bot \mid \top & \text{lock status}
\end{array}
$$

Table 1
Abstract syntax

entities executing in parallel are the named threads $n\langle t\rangle$, where t is the code being executed and n the name of the thread. The name n of the thread is at the same time the future reference under which the result value of t, if any, will be available. In this paper we restrict ourselves to the situation where the component consists of one object only, plus arbitrary many threads. A class $c[\![F, M]\!]$ carries a name c and defines its fields and methods in F and M. An object $o[c, F, L]$ with identity o keeps a reference to the class c it instantiates, stores the current value F of its fields, and maintains a *binary lock* L indicating whether any code is currently active inside the object (in which case the lock is taken) or not (in which case the lock is free). The symbols $\top$ and $\bot$ indicate that the lock is taken or free respectively.

The named threads $n\langle t\rangle$ are incarnations of method bodies "in execution". Each thread belongs to one specific object "inside" which it executes, i.e., whose instance variables it has access to. Built in object locks are used to rule out unprotected concurrent access to the object states: Though each object may have more than one method body incarnation partially evaluated, at each time point at most one of those bodies (the lock owner) can be active inside the object. The ν-operator is used for hiding and dynamic scoping, as known from the π-calculus.

Besides components, the grammar specifies the lower level syntactic constructs, in particular, methods, expressions, and (unnamed) threads, which are basically sequences of expressions. The further expressions claim, get, suspend, grab, and release deal with synchronization. They take care of releasing and acquiring the lock of an object appropriately. All of the features and their representation is pretty standard and (apart from the communication via method calls) not visible at the interface, we omit further details here and refer to the technical report [21].

2.2 Operational semantics

The operational semantics of a program being tested is given in two stages: steps *internal* to the program, and those occurring at the interface.

The internal rules deal with steps not interacting with the object's environment, such as sequential composition, conditionals, field look-up and update, etc. The rules are standard and we omit them here. More interesting and relevant are the "external" rules which describe the interaction of a component with its environment, by exchanging communication labels. The communication labels, the basic building blocks of the interface interactions, are given in Tab. 2. A component

or object exchanges information with the environment via *call*- and *return*-labels, and the interactions is either incoming or outgoing (marked ? resp. !). The label $n\langle call\ o.l(\boldsymbol{v})\rangle$ represents a call of method l in object o. In that label, n is a name identifying the thread that executes the method in the callee and is therefore the (future) reference under which the result of the method call will be available (if ever) for the caller. The incoming label $n\langle return(v)\rangle?$ hands the value from the corresponding call back to the object, which renders it ready to be read. Its counterpart, the outgoing return, passes the value to the environment. Besides that, labels can be prefixed by bindings of the form $\nu(n{:}T)$ which express freshness of the transmitted name, i.e., scope extrusion. .

The interface behavior is given by rules as those of Tab. 3 (we show 2 of the four rules, dealing with incoming communication, the missing 2 for outgoing communication are similar). The external steps are given as transitions of the form $\Xi \vdash C \xrightarrow{a} \acute{\Xi} \vdash \acute{C}$, where Ξ and $\acute{\Xi}$ represents the assumption/commitment contexts of C before and after the step, respectively. In particular, the context contains the identities of the objects and threads known so far, and the corresponding typing information. This information is *checked* in incoming communication steps, and updated when performing a step (input or output). These two operations are captured by the following notation

$$\Xi \vdash a : T \qquad \text{and} \qquad \Xi + a \tag{1}$$

which constitute part of the rule premises in Tab. 3. Intuitively, they mean the following: label a is well-formed and well-typed wrt. the information Ξ and refers to an asynchronous call which results in a value of type T. The right-hand notation of (1) extends the binding context Ξ by the bindings transmitted as part of label a appropriately. For lack of space, we omit the formal definitions here. Intuitively, they make sure that only well-typed communication can occur and that the context is kept up-to date during reduction. Rule CallI deals with incoming calls, and

γ	$::= n\langle call\ n.l(\boldsymbol{v})\rangle \mid n\langle \mathsf{return}(n)\rangle \mid \nu(n{:}T).\gamma$	basic labels
a	$::= \gamma? \mid \gamma!$	input and output labels

Table 2
Structured communication labels

$$\frac{a = \nu(\Xi').\ n\langle call\ o.l(\boldsymbol{v})\rangle? \quad \Xi \vdash a : T \qquad \acute{\Xi} = \Xi + a}{\Xi \vdash C \parallel o[c, F, \bot] \xrightarrow{a} \acute{\Xi} \vdash C \parallel o[c, F, \top] \parallel n\langle \mathsf{let}\ x{:}T = M.l(o)(\boldsymbol{v})\ \mathsf{in}\ \mathsf{release}(o); x\rangle}\ \textsc{CallI}$$

$$\frac{a = \nu(\Xi').\ n\langle return(v)\rangle? \qquad \Xi \vdash a : ok \qquad \acute{\Xi} = \Xi + a}{\Xi \vdash C \xrightarrow{a} \acute{\Xi} \vdash C \parallel n\langle v\rangle}\ \textsc{RetI}$$

Table 3
External steps

basically adds the new thread n (which at the same time represents the future reference for the eventual result) in parallel with the rest of the program. The notation $M.l(o)(\boldsymbol{v})$ represents the parameter passing of the actual values to the method body t, where s is the "self"-parameter, which is substituted by the identity o of the callee.

We write $\Xi_1 \vdash C_1 \stackrel{t}{\Longrightarrow} \Xi_2 \vdash C_2$ if $\Xi_1 \vdash C$ reduces in a number of internal and external steps to $\Xi_2 \vdash C_2$, exhibiting t as the trace of the external steps.

3 A behavioral interface specification language

The behavior of an object in a particular execution is, at the interface, described by a sequence of labels as given by Tab. 2. The black-box behavior of an object can therefore be described by a set of *traces*, each consisting of a finite sequence of labels. This would be the same also for a component consisting of a set of objects, for that matter. To specify sets of label traces, we employ a simple trace language with prefix, choice and recursion. Table 4 contains its syntax. The syntax of the labels in the specification language, naturally, quite resembles the labels of Tab. 2. Comparing Tabs. 2 and 4, there are two differences: first, instead of names or references n, the specification language here uses variables. Second, the labels here allow a binding of the form $(x{:}T).\gamma$, which has no analog in Tab. 2; the form $\nu(x{:}T).\gamma$ corresponds to $\nu(n{:}T).\gamma$, of course. Both binding constructs act as variable declarations, with the difference that $\nu(x{:}T).\gamma$ not just introduces a variable, but in addition asserts that the names represented by that variable must be fresh. The binding $(x{:}T).\gamma$ corresponds to a conventional variable declaration, introducing the variable x which represents arbitrary values (of type T), either fresh or already known.

The grammar given in Tab. 4 allows to specify sets of traces. Not all specifications, however, are meaningful, i.e., describe traces actually possible at the interface of a component. We rule out such ill-formed specifications by introducing restrictions on: *typing:* Values handed over must correspond to the expected types for that methods; *scoping:* Variables must be declared (together with their types) before their use; and *communication patterns:* No value can be returned before a matching outgoing call has been seen at the interface. In addition we must take care to consider the *polarity* of the specification. In the specification, it is important to distinguish between input and output interactions, as input messages are under the control of the environment, whereas the outputs are to be provided by the object as specified. This splits the specification into an *assumption* part under the responsibility of the environment, and a commitment part, controlled by the component. To specify non-deterministic behavior, the language supports a choice

γ	$::= x\langle call\ x.l(\boldsymbol{x})\rangle \mid x\langle \mathsf{return}(x)\rangle \mid \nu(x{:}T).\gamma \mid (x{:}T).\gamma$	basic labels
a	$::= \gamma? \mid \gamma!$	input and output labels
φ	$::= X \mid \epsilon \mid a\ .\ \varphi \mid \varphi + \varphi \mid \mathsf{rec}\ X.\varphi$	specifications

Table 4
Specification language

$$\frac{}{\nu(\Xi) \,.\, \gamma_1! \,.\, \gamma_2! \,.\, \varphi \equiv_{obs} \nu(\Xi) \,.\, \gamma_2! \,.\, \gamma_1! \,.\, \varphi}\ \textsc{Eq-Switch}$$

$$\frac{\vdash (\varphi_1 + \varphi_2) : wf^!}{\gamma! \,.\, (\varphi_1 + \varphi_2) \equiv_{obs} \gamma! \,.\, \varphi_1 + \gamma! \,.\, \varphi_2}\ \textsc{Eq-Plus} \qquad \mathsf{rec}\ X.\varphi \equiv_{obs} \varphi[\mathsf{rec}\ X.\varphi/X] \qquad \textsc{Eq-Rec}$$

Table 5
Observational equivalence

operator, and we distinguish between choices taken by the environment—external choice—and those the object is responsible for—internal choice. Especially, we do not allow so-called mixed choice. Cf. [21] for details about the formalization of these restrictions, presently just note that it is required that specifications are well-formed, and $\Xi \vdash \varphi : wf^p$ stands for the corresponding judgment. The metavariable p (for polarity) stands for either ?, !, or ?!, where ?! indicates the polarity for an empty sequence or for a process variable, and ? and ! indicate well-formed input and output specifications respectively.

3.1 Observational blur

Creol objects communicate asynchronously and the order of messages might not be preserved during communication. The order observed by an external observer or tester does not necessarily reflect the order in which the messages were sent, therefore an observed "wrong" order of communication should not be taken to be an error and we must relax the specification up-to some appropriate notion of *observational equivalence*, denoted by $\equiv_{obs}$ and defined by the rules of Tab. 5. Note that the purpose is not to reconstruct some "correct" order of communication. When testing a component, we control the communication, the test specification and framework plays the role of both environment (generating input to the CUT) and observer (controlling output), but want to retain the external perspective in order to test up-to observability. When testing a given object, we specify the order in which the inputs are consumed by the object, rather than the time they have been generated. In this way we specify the input scheduling of the object, which makes our specifications more expressive than in the case of blurring input. At the same time, we specify the outputs of the object as seen from the environment. We therefore blur the output, but not the input. This setting allows synchronous parallel composition. Input blur may be beneficial in other settings, and has e.g. been applied in a reasoning system for Creol based on Hoare logic [15]: In the presented compositional reasoning system, message generation is considered observable, but not messages consumption. Hence, in that system, input is blurred, but not output.

Rule Eq-Switch captures the asynchronous nature of communication, in that the order of outgoing communication does not play a role. The definition corresponds to the one given in [37] and also of [29], in the context of multi-threading concurrency. Rule Eq-Plus allows to distribute an output over a non-deterministic choice, *provided* that it is a choice over outputs, as required by the well-formed condition in the premise. Rule Eq-Rec finally expresses the standard unrolling of recursive definitions. The operational semantics of the specification language is

$$\frac{\Xi \vdash C \xrightarrow{\tau} \Xi \vdash \acute{C}}{\Xi \vdash C \parallel \varphi \rightarrow \Xi \vdash \acute{C} \parallel \varphi}\ \textsc{Par-Int} \qquad \frac{\vdash a \lesssim_\sigma b \quad \Xi_1 \vdash C \xrightarrow{a} \acute{\Xi}_1 \vdash \acute{C} \quad \Xi_1 \vdash \varphi \xrightarrow{b} \acute{\Xi}_2 \vdash \acute{\varphi}}{\Xi_1 \vdash C \parallel \varphi \rightarrow \acute{\Xi}_1 \vdash \acute{C} \parallel \acute{\varphi}\sigma}\ \textsc{Par}$$

$$\frac{\Xi \vdash \varphi : wf^?}{\Xi \vdash \nu(\Xi').(C \parallel n\langle \mathsf{let}\, x{:}T = o.l(\boldsymbol{v})\, \mathsf{in}\, t\rangle \parallel \varphi) \rightarrow \lightning}\ \textsc{Err-Call} \qquad \frac{\Xi \vdash \varphi : wf^?}{\Xi \vdash \nu(\Xi').(C \parallel n\langle v\rangle \parallel \varphi) \rightarrow \lightning}\ \textsc{Err-Ret}$$

Table 6
Parallel composition

straight forward reduction taking the rules of Tab. 5 into consideration (cf. [21]).

3.2 Asynchronous testing of objects

Next we put together the (external) behavior of an object (Sect. 2) and its intended behavior specified as in Sect. 3. Table 6 defines the interaction of the interface specification, φ, with the component, basically by synchronous parallel composition. Both φ and the component must engage in corresponding steps, which, for incoming communication schedules the order of interactions with the component whereas for outgoing communication the interaction will take place only if it matches an outgoing label in the specification and an error is raised if input is required by the specification. The component can proceed on its own via internal steps (cf. rule Par-Int). Rule Par requires that, in order to proceed, the component and the specification must engage in the "same" step, where φ's step b is matched against the step a of the component. Here $\vdash a \lesssim_\sigma b$ states that there exist a substitution σ such that the label a produced by the component and the label b specified by the interface description can be matched. Note that after a successful application of the Par rule, variables in the specification may have been substituted with concrete values. We omit the details of the matching and refer to the technical report [21]. The rules Err-Call and Err-Ret report an error if the specification requires an input as the next step and the object however could do an output, either a call or a return. In the rule $\lightning$ indicates the occurrence of an error. Note that the equivalence relation, according to the rule Eq-Switch, allows the reordering of outputs, but not of inputs.

4 A specification-driven interpreter for Creol

The operational semantics of Creol is formalized in rewriting logic [35] and executable on the Maude rewriting engine [9], this gives an interpreter for Creol. The executable framework for testing Creol components that we have implemented includes: our behavioral interface specification language formalized in rewriting logic and a modified version of the Creol interpreter. We obtain a *specification-driven interpreter* for testing by synchronizing the communication between specification terms and objects. Input to the component is generated non-deterministically within the bounds of the specification, and at the same time it is tested that the output behavior of the object conforms to the specification, the internal activity is

unmodified compared to the standard interpreter.

The default behavior for Creol is to place incoming method calls into the callee's input queue from which calls are non-deterministically selected for execution. For the specification-driven interpreter if an incoming call is specified and the lock of the object is free the corresponding method code should start executing immediately. In the implementation the incoming messages are generated directly from the specification.

The standard process of executing a Creol model in Maude consists of giving as input to Maude: an initial configuration and the interpreter for the Creol language and then let Maude rewrite this configuration. This allows for simulation of the model behavior. In addition one may use Maude's *search* command to search for specific result configurations. For testing a component, instead of using the initial model configuration as input to Maude we extract from the model one object together with its class definitions. This is the component under test (CUT). The CUT, its behavioral specification, and the modified interpreter for Creol is taken as input to Maude. Thus it is possible to test specific behavioral properties of selected objects in a large model.

A standard Creol state configuration (Cfg) is a multiset of objects, classes, and messages and the Maude rewrite rules for transitions are of the form `rl Cfg => Cfg'`. For the specification-driven interpreter, we introduce terms `Spec` for specifications and add rules on the form `(Spec || O) Cfg => (Spec' || O') Cfg'` to test the object `O` with respect to `Spec`, where `||` represents the synchronous parallel composition. Each rule evolves the state of a specification and the state of an object in a synchronized manner: an interaction only takes place when it matches a complementary label in the specification. E.g., the Par rule in Tab. 6 is implemented by several Maude rules for the different kinds of communication events that may occur. We refer the reader to [22] for some examples.

In the implementation, we define associative and commutative (AC) output prefixes by declaring the prefix operator to be AC in the cases where an output label is prefixed to an output specification. Together with a Maude rule that implements distribution over choice (the rule Eq-Plus above), this enables the testing framework to do testing up-to observational equivalence.

5 Experimental results

This section describes two series of experiments, using the implementation sketched in the previous section. The experiments demonstrate the usefulness of the approach: using AC rewriting may considerably reduce the resource consumption, when testing asynchronously communicating objects. AC rewriting significantly pays off in terms of time and the number of rewrites. With regards to the state space, the effects are not so definite.

The first example is tailor-made to show the effects for a simple component. The second example is an abstracted version of the "loan quote example" known from the area of enterprise application integration [26]. The examples also illustrate how to use the interface specification language for testing component behavior and how to employ model checking via the *search* command of Maude to also achieve *verification*

of a component with a trace specification. When using the search command, Maude not just explores *one* trace, but explores the set of behaviors given by the component together with the interface trace description. That the system in general explores a set of traces, as opposed to just one, has the following reasons: first, exploring a trace (trivially) means exploring all prefixes; that, of course, does not only apply to using Maude's search, but to simple rewriting as well. Second, the specification may contain non-determinism (besides the fact that also the component may behave non-deterministically). Finally, and most important in our context, one trace is always meant up-to the "observational blur", as specified in Tab. 5.

To measure the effect of AC rewriting, both series of experiments are carried out two times, either with AC rewriting switched on, or else off. When AC equivalence on the specification is switched off, we use an equivalent but expanded version of the specification to compare the results.

In the first example, the component under test consists of one object with n methods m_1 through m_n. The specification prescribes that all methods must have been called before any method may return. In Creol this is implemented by combining processor release points and *await* guards [31]. The behavioral specification for 3 methods reads:

$$\varphi_{c3} = n_1\langle call\ c.m_1(x_1)\rangle? \,.\, n_2\langle call\ c.m_2(x_2)\rangle? \,.\, n_3\langle call\ c.m_3(x_3)\rangle? \,. \\ (n_1\langle return(y_1)\rangle! \,.\, n_2\langle return(y_2)\rangle! \,.\, n_3\langle return(y_3)\rangle!) \,.\, \epsilon$$

A test is executed by giving the Maude command: `rew` (φ_{c3} `|| c) cClass .`, where c represents the Creol object. Maude rewrites the configuration, either resulting in an error reported when the component is about to execute an unspecified output, or stopping when no more rules apply. In the latter case, if the original specification is fully consumed this gives evidence that the component conforms to the specification, in the sense that test execution of c only leads to output foreseen by the specification φ_{c3}. This conformance relation is similar to the input-output conformance relation (**ioco**) of [40].

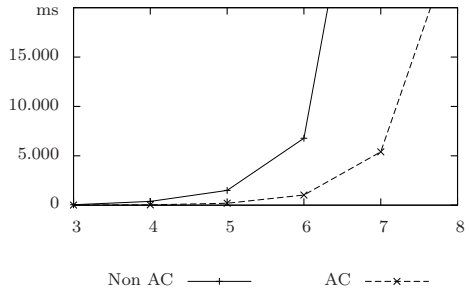
Definition 5.1 Let $out(\varphi\ after\ t)$ represent the set of all possible output events that is specified by φ after execution of the trace t. Let $out(c\ after\ t)$ represent the set of possible output events for the component c after execution of t. Let $traces(\varphi)$ be the set of traces that the specification designates. Our conformance relation *conf* is defined as follows:

$$c\ conf\ \varphi \Leftrightarrow_{def} \forall t \in traces(\varphi) :\ out(c\ after\ t) \subseteq out(\varphi\ after\ t)$$

Depending on the internal interleaving of the threads initiated by the method calls, different outcomes are possible. Maude's *search* command can be used to do a breadth first search for error configurations in the reachable state space:

```
search  in PROGRAM :  φc3 || c cClass =>+
                      φ || conf errorMsg(S:String) .
```

By altering the order of the input labels in the specification, we can easily check how different scheduling of input affect the execution of the object. E.g., a search for error states from a specification φ'_{c3} where the order of calls are to m_1, m_3, and



n	ms CPU time	
	AC	*Non AC*
3	16	47
4	38	379
5	198	1.498
6	1.030	6.782
7	5.407	49.311
8	27.894	NA
9	153.316	NA

Fig. 1. Verification of c with and without AC rewriting.

m_2 gives no solutions, which means that with the methods called in this order, the component cannot fail to conform to the specification.

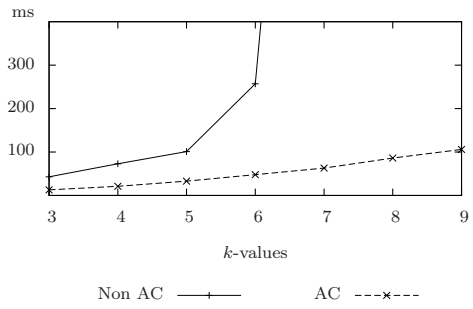
The two series of data, plotted in Fig. 1, show the time needed for exploring the state space with or without AC rewriting, where n is the number of methods. The figures show that with AC rewriting the increase in number of rewrites is considerably less than using the equivalent, expanded version of the specification.

In the second example, a *broker* acts as an intermediary between a client and several providers of some service (cf. [26]). Initially we consider a broker that after being requested to do so by a client queries a fixed number of providers for a (price) quote and returns an answer to the client giving the best alternative found. A specification for a broker querying two service providers can be given as:

$$\begin{aligned}\varphi_b = {} & n_{c1}\langle call\ b.getP(x)\rangle?\ . \\ & (\, n_1\langle call\ p_1.getQ(x)\rangle!\ .\ n_2\langle call\ p_2.getQ(x)\rangle!\,)\ . \\ & n_1\langle return(v_1)\rangle?\ .\ n_2\langle return(v_2)\rangle?\ .\ n_{c1}\langle return(v)\rangle!\ .\ \epsilon.\end{aligned}$$

Note that whereas the previous example illustrated generation of incoming calls to the component and testing of outgoing returns from the component, this example also includes testing of outgoing calls, and generation of incoming returns. For incoming returns, the test framework generates pseudo-random, type correct return values. For this specification a broker component would be non-conforming if it were to call the providers before receiving a call from the client and also if it were to return the initial call from the client before finishing its interaction with the providers.

In an open setting, the number of providers that a broker knows is likely to change over time, hence we assume that a broker will be notified by new providers and establish connections with them as well as losing connections with others. A further developed version of the broker supports this by allowing the client to give the number of providers that the broker must query before giving a response as a parameter to the call to the method *getP*. The method *getP* now takes two parameters, the name of the service for which a quote is requested, and the number of providers the broker should contact. To verify the behaviour of this new broker we



k	ms CPU time	
	AC	*Non AC*
3	13	43
4	21	73
5	33	101
6	48	257
7	63	1.965
8	86	17.796
9	106	NA

Fig. 2. Verification of the broker component

use a series of specifications on the following form

$$\begin{aligned}\varphi_{bk} = {} & n_{c1}\langle \mathit{call}\ b.\mathit{getP}(x,k)\rangle?\ .\\ & (\mathit{provider\ registration})\ .\\ & (\,n_1\langle \mathit{call}\ p_1.\mathit{getQ}(x)\rangle!\ .\ \ldots\ .\ n_k\langle \mathit{call}\ p_k.\mathit{getQ}(x)\rangle!\,)\ .\\ & n_1\langle \mathit{return}(v_1)\rangle?\ .\ \ldots\ .\ n_k\langle \mathit{return}(v_k)\rangle?\ .\ n_{c1}\langle \mathit{return}(v)\rangle!\ .\ \epsilon\ ,\end{aligned}$$

where k is the number of providers. Figure 2 plots the times of AC rewriting, resp. explicit rewriting against k. The experiments were carried out using Maude 2.4, on Red Hat Enterprise Linux 5.3 as operating system. The system was an AMD Athlon 64 X2 Dual Core Processor 4800+ with 1000 MHz cpu, 512 KB cache size, and 2 Gb memory.

6 Conclusion

We have presented a formalization of a concurrent object-oriented language and a behavioral specification language, for testing and verification of asynchronously communicating objects. Potential reorderings of communication events occur due to network properties. Our approach describes one way to deal with such situations, namely by defining rewriting specifications modulo AC for output events. One advantage of this approach is that we can define precisely the scheduling of input, and test internal synchronization properties of the object. When evaluating our approach by experimental case studies we get evidence that using modulo AC rewriting enable us to cover more extensive test cases than we could do otherwise.

Testing of Creol models is relevant also for testing of implementations in languages like C or Java: First indirectly, since many forms of non-determinism inherent in distributed system can be formalized by means of associativity and commutativity, our results are relevant also for other languages with asynchronous communication, and for alternative definitions of observational equivalence. Second, and more directly, in [24] and [1] it is shown how different testing techniques can be employed to check for conformance between a Creol model and an industrial distributed system implemented in C. In [24] the technique of dynamic symbolic execution is used to test for conformance between the Creol model and the implementation. Using the same case study, the authors of [1] show how to instrument

existing Creol models for testing. Aspect-C is used to insert event recording points into the existing code of the SuT. The model is likewise instrumented with synchronisation points. A tester process is used to replay the recorded events in the model and synchronises with events recorded by the tester, only allowing the model to proceed beyond synchronisation points if the corresponding event was recorded in the SuT. Thus conformance of implementation and Creol model may be verified. Combining these methods with our method for verification of conformance between the Creol model and the specification yields a method for conformance testing of implementations against a specification.

6.1 Related work

Systematic testing is indispensable to assure quality of software and systems. [7] presents an approach to integrate black-box and white-box testing for object-oriented programs. Equivalence is based on the idea of observably equivalent terms and fundamental pairs as test cases, but not in an asynchronous setting.

Godefroid et.al. [20] describe how state-space reductions can be achieved for *input* sequences in the context of constraint-based programming languages. A test algorithm is proposed which systematically generates all possible behavior by selecting input events non-deterministically from a predefined set. By exploiting the inability of constraint languages to observationally distinguish permutations of unordered sets of inputs, the combinatorial explosion is reduced, and a significantly more effective test algorithm is presented. A main difference from our approach is that the reduction in the state space is derived from the structure of the constraint-program itself and not from commutativity of the communicated events. The testing process is driven by the state-space exploration tool VeriSoft [19].

The paper [14] describes compositional analysis based on combining components with specifications. Also here VeriSoft is used for bounded model checking of assume/guarantee specifications, built-in partial order reduction contributes to efficiency of the analysis. However, both the object interaction model, shared variables, and the specifications, invariant based, using Hoare logic, differ from ours.

In [4] assumptions are used as environments to drive individual components for unit testing. LTSs are used to model the behavior of components. An interesting feature of this work, absent in ours, is techniques for automatic generation of exactly the assumptions that a component needs to make about the environment for some property to hold.

Testing for *concurrent* object-oriented programs based on synchronization sequences is investigated in [8], using Petri nets and OBJ as foundation. In his thesis [33], Long presents ConAn ("concurrency analyser"), which generates test drivers from test scripts. The method allows to specify sequences of component method calls and the order in which the calls should be issued (see also [34,38]). For scheduling the intended order, an external *clock* is used, introduced for the purpose of testing. The NModel-framework, comprehensively covered in [28], offers model-based analysis and model-based testing for $C^{\#}$, where abstract models, generally speaking transition systems, of object-oriented programs are used for testing. Related and likewise developed at Microsoft is the Spec Explorer approach (and its prede-

cessor AsmLT), a tool for testing reactive, object-oriented programs. Underlying the model programs, given e.g., in the *Spec#* specification language, are "model automata" which can be seen as a combination of interface automata and abstract state machines (ASMs), and which are used for test case generation. Dealing with non-determinism, the models separate observable and controllable actions, similar as we distinguish between inputs and output actions in our specification language. Relying on game theoretic foundations, their notion of conformance is based on alternating simulation, not on comparing traces, as in this work. To cope with large and potentially infinite state spaces, Spec Explorer uses different abstraction and pruning techniques. One is based on building a quotient of the model automaton by identifying states which are considered equivalent ("state groupings", cf. [23] and [6]). These state groupings correspond to predicate abstraction known from model checking and serve a similar purpose as the observable equivalence presented here. I.e., the are used to reduce the state space, but are user-given and not specifically capturing observably equivalent states due to asynchronous communication. For a thorough discussion of Spec Explorer and links to further results in that context, see [42].

Another well-established approach for functional testing is input/output conformance testing (ioco for short) [39,40]. Ioco is based on input-output transition systems, our conformance relation is closely related. Component-based testing and testing in context, using the ioco test theory, are studied in [41]. A number of test-tools are based on variants of the ioco test theory, such as TGV, TestGen, and TorX. In the context of ioco testing, [17] uses *symbolic* transition systems to counter the state explosion problem. Unit testing framework for actors, i.e., active concurrent objects, is presented in [11], using the discrete event based simulation environment OPNET. Validation of component interfaces specified in rewriting logic is the subject also of [30]. [36] considers Creol and investigates how different scheduling of object activity restrict the behavior. The focus is on *intra*-object scheduling, and on test purposes as assertions on the *internal* state of the object. This is in contrast to our focus on the interface communication.

6.2 Future work

Creol has successfully been used to model complex and highly dynamic communication systems, e.g. wireless sensor networks in [32], where the Ad hoc On-Demand Distance Vector (AODV) routing algorithm is used as a case study. ASK is an industrial size multi-threaded, asynchronous application for connecting people. A substantial part of ASK has been modelled in Creol [1]. Both these models are complex. The similarity of Creol and an object-oriented programming language, and Creol's expressiveness allow for models that are structurally close to the AODV algorithm resp. the ASK system itself. This leads to a need for testing the models. We are currently working on applying our method for model-based testing of Creol models to the AODV model.

Acknowledgement
We thank Rudolf Schlatte for insight into application testing with Creol. And we thank the anonymous referees for constructive criticism and hints to related work.

References

[1] Bernhard Aichernig, Andreas Griesmayer, Rudolf Schlatte, and Andries Stam. Modeling and testing multi-threaded asynchronous systems with Creol. In *Proceedings of the 2nd International Workshop on Harnessing Theories for Tool Support in Software (TTSS'08)*, ENTCS. Elsevier, 2008.

[2] A. Belinfante, J. Feenstra, R. d. Vries, J. Tretmans, N. Goga, L. Feijs, S. Mauw, and L. Heerink. Formal test automation: A simple experiment. In G. Csopaki, S Dibuz, and K. Tarnay, editors, *Proceedings of the 12th International Workshop on Testing of Communicating Systems*, pages 179–196. Kluwer Academic Publishers, 1999.

[3] G. Bernot. Testing against formal specification: A theoretical view. In Samson Abramsky and T. S. E. Maibaum, editors, *TAPSOFT '91, Volume 1*, volume 493 of *Lecture Notes in Computer Science*, pages 99–119. Springer-Verlag, 1991.

[4] C. Blundell, D. Giannakopoulou, and C. S. Pasareanu. Assume-guarantee testing. In *Proceedings of SAVCBS'05*, pages 7–14, 2005.

[5] M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, and A. Pretschner, editors. *Model-Based Testing of Reactive Systems*, volume 3472 of *Lecture Notes in Computer Science*. Springer-Verlag, 2005.

[6] Colin Campbell and Margus Veanes. State exploration with multiple state groupings. In *12th International Workshop in Abstratc State Machines, ASM'05*. Laboratory of Algorithms, Complexity, and Logic, University Paris 12, 2005.

[7] H. Y. Chen, T. H. Tse, F. T. Chan, and T. Y. Chen. In black and white: An integrated approach to class-level testing of object-oriented program. *ACM Transactions of Software Engineering and Methodology*, 7(3):250–295, 1998.

[8] Huo Yan Chen, Yu Xia Sun, and T. H. Tse. A strategy for selecting synchronization sequences to test concurrent object-oriented software. In *Proceedings of the 27th International Computer Software and Application Conference (COMPSAC 2003), Los Angeles, California*. IEEE Computer Science Press, 2003.

[9] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. The Maude 2.0 system. In Robert Nieuwenhuis, editor, *Proceedings of the 14th International Conference on Rewriting Techniques and Applications (RTA 2003)*, volume 2706 of *Lecture Notes in Computer Science*, pages 76–87. Springer-Verlag, June 2003.

[10] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. *The Maude Manual (version 2.1.1)*. SRI International, Menlo Park, April 2005.

[11] Mark E. Coyne, Scott R. Graham, Kenneth M. Hopkinson, and Stuart H. Kurkowski. A methodology for unit testing actors in proprietary discrete event based simulations. In *WSC '08: Proceedings of the 40th Conference on Winter Simulation*, pages 1012–1019. Winter Simulation Conference, 2008.

[12] The Creol language. http://heim.ifi.uio.no/creol, 2007.

[13] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz. Model-based testing in practice. In *Proceedings of the 1999 International Conference on Software Engineering, 1999*, pages 285–294, 1999.

[14] Juergen Dingel. Computer-assisted assume/guarantee reasoning with verisoft. In *25th International Conference on Software Engineering (ICSE'03)*, 2003.

[15] Johan Dovland, Einar Broch Johnsen, and Olaf Owe. Observable behavior of dynamic systems: Component reasoning for concurrent objects. In Dina Goldin and Farhad Arbab, editors, *Proc. Workshop on the Foundations of Interactive Computation (FInCo'07)*, volume 203 of *Electronic Notes in Theoretical Computer Science*, pages 19–34. Elsevier, May 2008.

[16] Jean-Claude Fernandez, Claude Jard, Thierry Jéron, and César Viho. An experiment in automatic generation of test suites for protocols with verification technology. *Science of Computer Programming*, 29(1–2):123–146, July 1997.

[17] Lars Frantzen, Jan Tretmans, and Tim A. C. Willemse. Test generation based on symbolic specifications. In J. Grabowski and B. Nielsen, editors, *Third International Workshop on Formal Approaches to Testing of Software, FATES 2004*, pages 1–15, 2005.

[18] Marie-Claude Gaudel. Testing can be formal, too. In Peter D. Mosses, Mogens Nielsen, and Michael I. Schwarzbach, editors, *Proceedings of TAPSOFT '95*, volume 915 of *Lecture Notes in Computer Science*, pages 82–96. Springer-Verlag, 1995.

[19] Patrice Godefroid. Model checking for programming languages using VeriSoft. In *Proceedings of POPL '97*, pages 174–186. ACM, January 1997.

[20] Patrice Godefroid, L. Jagadeesan, R. Jagadeesan, and K. Läufer. Automated systematic testing for constraint-based interactive services. In *Proceedings of the 8th International Symposium on the Foundations of Software Engineering (FSE' 2000)*. ACM, 2000.

[21] Immo Grabe, Marcel Kyas, Martin Steffen, and Arild Braathen Torjusen. Executable interface specifications for testing asynchronous Creol components. Technical Report 375, University of Oslo, Dept. of Computer Science, July 2008.

[22] Immo Grabe, Martin Steffen, and Arild Braathen Torjusen. Executable interface specifications for testing asynchronous Creol components. In *Proceedings of the 3rd International Conference on Fundamentals of Software Engineering (FSEN'09), 15 - 17 April, Kish Island, Persian Gulf*, Lecture Notes in Computer Science. Springer-Verlag, 2010. To appear.

[23] W. Grieskamp, Y. Gurevitch, W. Schulte, and M. Veanes. Generating finite state machines for abstract state machines. In *Proceedings of ISSTA'07, volume 27 of Software Engineering Notes*, pages 112–122. ACM, 2002.

[24] Andreas Griesmayer, Bernhard Aichernig, Einar Broch Johnsen, and Rudolf Schlatte. Dynamic symbolic execution of distributed concurrent objects. In Arnd Poetzsch-Heffter David Lee, Antónia Lopes, editor, *Proceedings of FMOODS/FORTE'09*, volume 5522 of *Lecture Notes in Computer Science*, pages 225–230. Springer-Verlag, June 2009.

[25] J. He and K. Turner. Protocol-inspired hardware testing. In G. Csopaki, S Dibuz, and K. Tarnay, editors, *Proceedings of the 12th International Workshop on Testing of Communicating Systems*, pages 131–147. Kluwer Academic Publishers, 1999.

[26] Gregor Hohpe and Bobby Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley, 2003.

[27] International Telecommunication Union. Open Distributed Processing - Reference Model parts 1–4. Technical report, ISO/IEC, Geneva, July 1995.

[28] Jonathan Jacky, Margus Veanes, Colin Campbell, and Wolfram Schulte. *Model-Based Software Testing and Analysis with C♯*. Cambridge University Press, 2008.

[29] Alan Jeffrey and Julian Rathke. A fully abstract may testing semantics for concurrent objects. In *Proceedings of LICS '02*, pages 101–112. IEEE, Computer Society Press, July 2002.

[30] Einar Broch Johnsen, Olaf Owe, and Arild B. Torjusen. Validating behavioral component interfaces in rewriting logic. *Fundamenta Informaticae*, 82(4):341–359, 2008.

[31] Einar Broch Johnsen, Olaf Owe, and Ingrid Chieh Yu. Creol: A type-safe object-oriented model for distributed concurrent systems. *Theoretical Computer Science*, 365(1–2):23–66, November 2006.

[32] Wolfgang Leister and Joakim Bjørk. Modelling routing algorithms for wireless sensor networks in Creol, 2009. Presented at the 21st Nordic Workshop on Programming Theory, NWPT '09, Copenhagen.

[33] Bradley Long. *Testing Concurrent Java Components*. PhD thesis, University of Queensland, July 2005.

[34] Bradley Long, D. Hofmann, and P. Strooper. Tool support for testing concurrent Java components. *IEEE Transactions on Software Engineering*, 29(6):555–566, 2003.

[35] José Meseguer. Conditional rewriting as a unified model of concurrency. *Theoretical Computer Science*, pages 73–155, 1992.

[36] Rudolf Schlatte, Bernhard Aichernig, Frank de Boer, Andreas Griesmayer, and Einar Broch Johnsen. Testing concurrent objects with application-specific schedulers. In John Fitzgerald and Anne Haxthausen, editors, *International Colloquium on Theoretical Aspects of Computing (ICTAC)*, volume 5160 of *LNCS*. Springer, 2008.

[37] Martin Steffen. *Object-Connectivity and Observability for Class-Based, Object-Oriented Languages*. Habilitation thesis, Technische Faktultät der Christian-Albrechts-Universität zu Kiel, July 2006.

[38] Paul Strooper and Luke Wildman. Testing concurrent Java components. In *ICSE COMPANION '07: Companion to the proceedings of the 29th International Conference on Software Engineering*, pages 161–162, Washington, DC, USA, 2007. IEEE Computer Society.

[39] Jan Tretmans. Test generation with inputs, outputs, and quiescence. In T. Margaria and B. Steffen, editors, *Proceedings of TACAS '96*, volume 1055 of *Lecture Notes in Computer Science*, pages 127–146. Springer-Verlag, 1996.

[40] Jan Tretmans. Test generation with inputs, outputs, and repetetive quiescence. *Software — Concepts and Tools*, 17(3):103–120, 1996.

[41] Machiel van Bijl, Arend Rensink, and Jan Tretmans. Compositional testing with ioco. In *TestCom/FATES 2003*, volume 2931 of *Lecture Notes in Computer Science*, pages 86–100. Springer-Verlag, 2003.

[42] Margus Veanes, Colin Campbell, Wolfgang Grieskamp, Wolfram Schulte, Nikolai Tillmann, and Lev Nachmanson. Model-based testing of object-oriented reactive systems with spec explorer. In *Formal Methods and Testing*, volume 4949 of *Lecture Notes in Computer Science*, pages 39–76. Springer-Verlag, 2008.

Part III

Appendix

A-1 A specification-driven interpreter for Creol

```
  in creol-interpreter .
fmod SPEC-DATATYPES is
  pr CREOL-DATATYPES .
  sort BaseLab .
  sorts BaseCall BaseRet .
  subsorts BaseCall BaseRet < BaseLab .
  sort CommLab .
  sorts InLab OutLab .
  subsorts InLab OutLab < CommLab .
  sorts In Out Spec .
  subsorts In Out < Spec .
  sort SpecVar .
  sorts  SpecVarI SpecVarO .
  subsort SpecVarI < In .
  subsort SpecVarO < Out .
  subsorts SpecVarI SpecVarO < SpecVar .
endfm

fmod SPEC-SYNTAX is
pr SPEC-DATATYPES .
  op call : Expr Expr String ExprList -> BaseCall [ctor] .
  op ret : Expr ExprList -> BaseRet [ctor] .
  op _! : BaseLab -> OutLab [ctor] .
  op _? : BaseLab -> InLab [ctor] .
  op epsIn : -> In .
  op epsOut : -> Out .
  op errSp : -> Spec .
  op eps : -> Spec .
  op _+_ : In In -> In [ctor comm prec 47 format (o ssb! sso o)  ] .
  op _+_ : Out Out -> Out [ctor comm prec 47 format (o ssr! sso o)  ] .
  op Xo : Nat  -> SpecVarO [ctor] .
  op Xi : Nat  -> SpecVarI [ctor] .
  op rec : SpecVar In  -> In .
  op rec : SpecVar Out  -> Out .
endfm

fmod SPEC-REORDER is
  pr SPEC-SYNTAX .
  sort OutPrefix .
  subsort OutLab < OutPrefix .
  op _._ : InLab Spec -> In [ctor prec 45 gather(e E)] .
  op _._ : OutPrefix In -> Out [ctor prec 45 gather(e E) ] .
  op _._ : OutPrefix Out -> Out [ctor prec 45 gather(e E) ] .
  *** Uncomment below to switch on/off AC for outputprefix
  *** 4 AC OutPrefix
  op _._ : OutPrefix OutPrefix -> OutPrefix [ctor assoc comm prec 45 ] .
  *** 4 NON AC OutPrefix, right associative.
  ***  op _._ : OutPrefix OutPrefix -> OutPrefix [ctor prec 45 gather(e E) ] .
endfm
```

```
fmod INTERFACE-SPEC  is
    pr SPEC-REORDER .
endfm

mod CREOL-SCHEDULER-UTILS is
  pr INTERFACE-SPEC .
  pr CREOL-SIMULATOR .

  vars D D' : Data . var DL : DataList . vars E  Tid Rcv R  : Expr .
  vars EL Args   : ExprList .  var Lab : Label .  var M : String .
  vars N N' CT : Nat . vars O O' Sender SO  : Oid . var PR : ProcRes .
  var Q : String . var a : CommLab . vars g g' : BaseLab . var msg : Msg .
  var oLab : OutLab . vars opf opf'  : OutPrefix . vars sp sp' : Spec .
  var subRes : DataSubst .  vars subst subst' S : Subst . var xV : SpecVar .

  op length : Spec -> Nat .
  op length : OutPrefix -> Nat .
  eq length(epsIn) = 0 .
  eq length(epsOut) = 0 .
  eq length(sp + sp') = max(length(sp),length(sp')) .
  eq length(rec(xV,sp)) = 999 .
  eq length(a . sp) = 1 + length(sp) .
  eq length(opf . sp) = length(opf) + length(sp) .
  eq length(opf . opf') = length(opf) + length(opf') .
  eq length(oLab) = 1 .
  op genMsg : Oid CommLab -> Msg .
  op genMsg : CommLab -> Msg .
  op procLab : Oid Nat CommLab -> ProcRes .

  sort ProcRes .
  op pRes : Msg Subst -> ProcRes .
  op getM : ProcRes -> Msg .
  op getS : ProcRes -> Subst .
  eq getM(pRes(msg,subst)) = msg .
  eq getS(pRes(msg,subst)) = subst .

  sort DataSubst .
  op dSub : Data Subst -> DataSubst .
  op getData : DataSubst -> Data .
  op getS : DataSubst  -> Subst .
  eq getData(dSub(D,subst)) = D .
  eq getS(dSub(D,subst)) = subst .

  sort EDPair .
  sort EDPairList .
  subsort EDPair < EDPairList .
  var edp : EDPair .
  var edps : EDPairList .
  op edPair : Expr Data -> EDPair [ctor].
  op edNoMatch : -> EDPair [ctor].
  op noedPair : -> EDPairList [ctor] .
  op __ : EDPairList EDPairList -> EDPairList [ctor assoc id: noedPair] .
```

```
op zip : ExprList DataList -> EDPairList .
eq zip(E :: EL,D :: DL) = edPair(E,D) zip(EL,DL) .
eq zip(emp,emp) = noedPair .
op match : EDPairList Subst  -> Subst .
eq match(noedPair,S) = S .
eq match(edPair(D,D'),S) = if D == D' then S else misMatch fi .
eq match(edPair(E,D),S) =
  if  $hasMapping(S,E) then
      if S[E] == D then S else misMatch fi
   else
    insert(E,D,S) fi .
eq match(edPair(E,D) edps,S) = match(edps,(match(edPair(E,D),S))) .
op getRcv : Oid Expr -> Oid .
eq getRcv(SO,D) = D .
eq getRcv(SO,E) = SO .
op getInvocLabel : Nat Expr -> Label .
eq getInvocLabel(CT,Lab) = Lab .
eq getInvocLabel(CT,E) = label(ob("EnvObj"),CT) .
op getRetLabel : Oid Nat Expr -> Label .
eq getRetLabel(O,CT,Lab) = Lab .
eq getRetLabel(O,CT,E) = label(O,CT) .
op getVals : Nat ExprList -> DataList .
eq getVals(CT,DL) = DL .
eq getVals(CT,EL) = $getVals(CT,EL,0) .
op $getVals : Nat ExprList Nat  -> DataList .
eq $getVals(CT,emp,N) = emp .
eq $getVals(CT,E :: EL,N) = genVal(CT,E,N) :: $getVals(CT + 1,EL,N + 1) .
op genVal : Nat Expr Nat -> Data .
eq genVal(CT,E,N) = int((trunc((random(CT) / 4294967295) * 9) + 1)) .
op getArgs : Nat Oid String ExprList -> DataList .
eq getArgs(CT,O,M,DL) = DL .
eq getArgs(CT,O,M,EL) = $getArgs(CT,EL,0) .
op $getArgs : Nat ExprList Nat  -> DataList .
eq $getArgs(CT,emp,N) = emp .
eq $getArgs(CT,E :: EL,N) = genArg(CT,E,N) :: $getArgs(CT + 1,EL,N + 1) .
op genArg : Nat Expr Nat -> Data .
eq genArg(CT,E,N) = str("Arg" + string((N + CT),10)) .

ceq [gen-invoc-msg] :
    procLab(O,CT,call(Tid,R,M,Args) ?)
    =
    pRes((invoc(Sender,Lab,M,DL) from Sender to Rcv),subst)
    if Rcv := getRcv(O,R)
       Lab := getInvocLabel(CT,Tid) /\
       Sender := caller(Lab) /\
       DL := getArgs(CT,Rcv,M,Args) /\
       subst := match((edPair(Tid,Lab) edPair(R,Rcv) zip(Args,DL)),noSubst) .

ceq [gen-ret-msg] :
  procLab(O,CT,ret(Tid,Args) ?)
    =
    pRes((comp(Lab,DL) from Sender to Rcv),subst)
    if Lab := getRetLabel(O,CT,Tid) /\
```

```
        Rcv := caller(Lab) /\
        Sender := ob("EnvObject") /\
        DL := getVals(CT,Args) /\
      subst := match(edPair(Tid,Lab) zip(Args,DL),noSubst) .

  op misMatch : -> Subst [ctor] .
  op noMismatch : Subst -> Bool .
  eq noMismatch(noSubst) = true .
  eq noMismatch((misMatch,subst)) = false .
  eq noMismatch((subst,subst')) = true [owise] .
  op app : Subst In -> In .
  op app : Subst Out -> Out .
  op app : Subst BaseLab -> BaseLab .
  op app : Subst OutPrefix -> OutPrefix .
  eq app(S,(g !))  = app(S,g) ! .
  eq app(S,(g ! . g' !))  = (app(S,g) ! . app(S,g') !) .
  eq app(S,(g ! . opf) . sp)  = (app(S,g) ! . app(S,opf)) . app(S,sp) .
  eq app(S,(g ! . opf))  = (app(S,g) ! . app(S,opf)) .
  eq app(noSubst,sp) = sp .
  eq app(S,epsIn) = epsIn .
  eq app(S,epsOut) = epsOut .
  eq app(S,g ! . sp) = app(S,g) ! . app(S,sp) .
  eq app(S,g ? . sp) = app(S,g) ? . app(S,sp) .
  eq app(S,sp + sp') =  app(S,sp) + app(S,sp') .
  eq app(S,rec(xV,sp)) = rec(xV,app(S,sp)) .
  eq app(S,call(Tid,R,M,Args)) = call(subE(S,Tid),subE(S,R),M,subE(S,Args)) .
  eq app(S,ret(Tid,EL)) = ret(subE(S,Tid),subE(S,EL)) .
  op subE : Subst ExprList ->  Data .
  eq subE(S,D) = D  [label subE1] .
  eq subE(S,E) = if $hasMapping(S,E) then S[E] else E fi .
  eq subE(S,E :: EL) = subE(S,E) :: subE(S,EL) .
  eq subE(S,emp) = emp .
  op someRet : -> BaseRet [ctor] .
  op someCall : -> BaseCall [ctor] .
  op matchCall : Label Data String DataList BaseCall -> Subst .
  eq matchCall(Lab,Rcv,Q,Args,someCall) = noSubst .

  ceq matchCall(Lab,Rcv,Q,DL,call(Tid,R,M,Args)) =
   if (M == Q and-then subst =/= misMatch) then subst else misMatch fi
   if subst := match(edPair(Tid,Lab)  edPair(R,Rcv) zip(Args,DL),noSubst) .

  op match : Expr Label -> Bool .
  eq match(label(O,N), label(O,N)) = true .
  eq match(E, Lab)  = false [owise] .
  op matchRet : Label DataList BaseRet -> Subst .
  eq matchRet(Lab,DL,someRet) = noSubst .

  ceq matchRet(Lab,DL,ret(Tid,Args)) =
   if (subst =/= misMatch) then subst else misMatch fi
   if subst := match(edPair(Tid,Lab) zip(Args,DL),noSubst) .
endm
```

```
mod CREOL-SCHEDULER is
  pr CREOL-SCHEDULER-UTILS .

  var A : Vid . var C  : String . var Cfg : Configuration . vars E Tid Rcv R  : Expr .
  vars EL Ps Args : ExprList . var LS : Labels . var Lab : Label . vars M Q : String .
  var MM : MMsg . var Msg : Msg . vars N CT : Nat . vars O SO   : Oid .
  var P : Process . var Res : ProcRes . var SL  : StmList . vars Subst S L  : Subst .
  var W : MProc . var a : CommLab . var acall : BaseCall . var aret : BaseRet .
  var g  : BaseLab . vars inSp inSp' : In . vars opfi opf opf' : OutPrefix .
  vars outSp outSp' : Out . vars sp sp' : Spec . var spS   : SpecSubst .
  var spSS  : SpecSubstSet . var xV : SpecVar .

  sort SynchConfig . sort SpecContext . sort SpecSubst .sort SpecSubstSet .
  subsort SpecSubst < SpecSubstSet .

  op <_>(_,_) : Spec Oid Nat   -> SpecContext .
  op _||_ : SpecContext Configuration  -> SynchConfig [ctor format (nn nn n d) ] .
  op debugMsg : String Universal -> Configuration [ctor poly (2) format (nnb onn) ] .
  op errorMsg : String -> Configuration [ctor format (nnr! o) ] .
  op cfgCnt : Nat -> Configuration [ctor] .
  op [_/_] : Spec SpecVar -> SpecSubst .
  op noSpSub : -> SpecSubstSet [ctor] .
  op __ : SpecSubstSet SpecSubstSet -> SpecSubstSet [ctor assoc comm id: noSpSub ] .
  eq spS spS = spS .
  op _{_} : In SpecSubstSet  -> In [ctor format (o nm! o m! no) ] .
  op _{_} : Out SpecSubstSet  -> Out [ctor format (o nm! o m! no) ] .
  rl [Eq-Rec] :  < rec(xV,sp) >(O,CT) => < sp{[rec(xV,sp) / xV]}  >(O,CT) .

  rl [Eq-Rec-Sub] : < rec(xV,sp){ spSS } >(O,CT) =>
                                              < sp{ [rec(xV,sp) / xV] spSS } >(O,CT) .

  rl [subst-branch] < (sp + sp'){ spSS } >(O,CT) => < sp{ spSS } + sp'{ spSS } >(O,CT) .
  rl [subst-epsIn] < epsIn{ spSS } >(O,CT) => < epsIn >(O,CT) .
  rl [subst-epsOut] < epsOut{ spSS } >(O,CT) => < epsOut >(O,CT) .
  rl [subst-var]  < xV{[sp / xV] spSS} >(O,CT) => < sp{ spSS } >(O,CT) .

  rl [subst-prefix-standard] < (a . sp){ spSS } >(O,CT) =>
                                              < a . (sp { spSS })  >(O,CT) .

  rl [subst-prefix-reorder] < ((opf) . sp){ spSS } >(O,CT) =>
                                              < (opf) . (sp { spSS })  >(O,CT) .

  rl [output-choice] : < outSp + outSp' >(O,CT) => < outSp' >(O,CT) .
  rl [input-choice] :  < inSp + inSp' >(O,CT) => < inSp' >(O,CT) .
  eq app(S,sp { spSS }) = app(S,sp){ spSS } .
  eq app(S,xV) = xV .
  eq opfi . (opf . sp + opf' . sp') =   (opfi . opf) . sp + (opfi . opf') . sp' .
```

```
*** OPERATIONAL RULES
crl [local-async-call] :
  < O : C | Att: S,Pr: { L | call(A ; E ; Q ; EL); SL },PrQ: W,
            Dealloc: LS,Ev: MM,Lcnt: N >
  =>
  < O : C | Att: S,Pr: { insert(A,label(O,N),L) | SL },PrQ: W,
            Dealloc: LS,Ev: MM,Lcnt: (N + 1) >
  invoc(O,label(O,N),Q,evalGuardList(EL,(S :: L),noMsg))
       from O to evalGuard(E,(S :: L),noMsg)
  if E == "this" .

crl [local-return] :
  < O : C |  Att: S,Pr: { L | return(EL); SL },PrQ: W,
             Dealloc: LS,Ev: MM,Lcnt: N >
  =>
  < O : C |  Att: S,Pr: { L | SL },PrQ: W,Dealloc: LS,Ev: MM,Lcnt: N >
  comp(L[".label"],evalGuardList(EL,(S :: L),noMsg)) from O to caller(L[".label"])
  if caller(L[".label"]) == O .


crl [PAR-remote-async-call] :
  < call(Tid,R,M,Ps) ! . sp >(O,CT) ||
  < O : C | Att: S,Pr: { L | call(A ; E ; Q ; EL); SL },PrQ: W,
            Dealloc: LS,Ev: MM,Lcnt: N > Cfg
  =>
  < app(Subst, sp) >(O,CT)  ||
  < O : C | Att: S,Pr: { insert(A,Lab,L) | SL },PrQ: W,
            Dealloc: LS,Ev: MM,Lcnt: (N + 1) > Cfg
  (invoc(O,Lab, Q,Args) from O to Rcv)
   if Lab := label(O,N)  /\
     Args := evalGuardList(EL,(S :: L),noMsg) /\
     Rcv := evalGuard(E,(S :: L),noMsg) /\
     Subst := matchCall(Lab,Rcv,Q,Args,call(Tid,R,M,Ps))  /\ noMismatch(Subst) .

crl [PAR-return] :
  < (aret ! . sp)  >(O,CT)  ||
  < O : C |  Att: S,Pr: { L | return(EL); SL },PrQ: W,Dealloc: LS,Ev: MM,Lcnt: N > Cfg
  =>
  < app(Subst,sp) >(O,CT)  ||
  < O : C |  Att: S,Pr: { L | SL },PrQ: W,Dealloc: LS,Ev: MM,Lcnt: N > Cfg
  comp(Lab,Args) from O to caller(L[".label"])
  if Lab := L[".label"] /\
     Args := evalGuardList(EL,(S :: L),noMsg) /\
      Subst := matchRet(Lab,Args,aret) /\ noMismatch(Subst) .

crl [PAR-incoming-call] :
  < acall ? . sp >(O,CT) ||
  < O : C | Att: S,Pr: idle,PrQ: W,Dealloc: LS,Ev: MM,Lcnt: N > Cfg
  =>
  < app(Subst,sp) >(O,CT + 1)  ||
  getM(Res) < O : C | Att: S,Pr: synch,PrQ: W,Dealloc: LS,Ev: MM,Lcnt: N > Cfg
  if Res := procLab(O,CT,acall ?) /\
     Subst := getS(Res) /\ noMismatch(Subst) .
```

```
crl [PAR-incoming-ret] :
  < (aret ? . sp) >(O,CT) ||
  < O : C | Att: S,Pr: P,PrQ: W,Dealloc: LS,Ev: MM,Lcnt: N > Cfg
  =>
  < app(Subst,sp) >(O,CT + 1) ||
  getM(Res) < O : C | Att: S,Pr: P,PrQ: W,Dealloc: LS,Ev: MM,Lcnt: N > Cfg
  if Res := procLab(O,CT,aret ?) /\
     Subst := getS(Res) /\ noMismatch(Subst) .

crl [PAR-ERROR] :
  < inSp >(O,CT)  ||
  < O : C | Att: S,Pr: { L | call(A ; E ; Q ; EL); SL },PrQ: W,
            Dealloc: LS,Ev: MM,Lcnt: N > Cfg
  =>
  < errSp >(O,CT)
  ||
  errorMsg("ERROR: Spec. requires input and the next statement is a call out. ")
  < O : C | Att: S,Pr: { L | call(A ; E ; Q ; EL); SL },PrQ: W,
            Dealloc: LS,Ev: MM,Lcnt: N >
  Cfg
  if E =/= "this" .

rl [PAR-ERROR-RETURN] :
  < inSp >(O,CT)  ||
  < O : C |  Att: S,Pr: { L | return(EL); SL },PrQ: W,
             Dealloc: LS,Ev: MM,Lcnt: N > Cfg
  =>
  < errSp >(O,CT) ||
  errorMsg("ERROR: Spec. requires input and the next statement is a return out.")
  < O : C |  Att: S,Pr: { L | return(EL); SL },PrQ: W,Dealloc: LS,Ev: MM,Lcnt: N >
  Cfg .

crl [PAR-call-out-opf] :
  < (acall ! . opf) . sp >(O,CT)  ||
  < O : C | Att: S,Pr: { L | call(A ; E ; Q ; EL); SL },PrQ: W,
            Dealloc: LS,Ev: MM,Lcnt: N > Cfg
  =>
  < app(Subst, (opf) . sp) >(O,CT)  ||
  < O : C | Att: S,Pr: { insert(A,label(O,N),L) | SL },PrQ: W,
            Dealloc: LS,Ev: MM,Lcnt: (N + 1) > Cfg
    (invoc(O,Lab, Q,Args) from O to Rcv)
    if Lab := label(O,N)  /\
      Args := evalGuardList(EL,(S :: L),noMsg) /\
      Rcv := evalGuard(E,(S :: L),noMsg) /\
      Subst := matchCall(Lab,Rcv,Q,Args,acall) /\ noMismatch(Subst) .
```

```
crl [PAR-return-opf] :
  < (aret ! . opf) . sp   >(O,CT)  ||
  < O : C |  Att: S,Pr: { L | return(EL); SL },PrQ: W,Dealloc: LS,Ev: MM,Lcnt: N >
  Cfg
  =>
  < app(Subst,((opf) . sp)) >(O,CT)
  ||
  < O : C |  Att: S,Pr: { L | SL },PrQ: W,Dealloc: LS,Ev: MM,Lcnt: N >
  Cfg
  comp(Lab,Args) from O to caller(L[".label"])
  if Lab := L[".label"] /\
     Args := evalGuardList(EL,(S :: L),noMsg) /\
     Subst := matchRet(Lab,Args,aret) /\ noMismatch(Subst) .
endm
eof
```